

The L^AT_EX3 Sources

The L^AT_EX Project*

Released 2026-03-16

Abstract

This is the typset sources for the `expl3` programming environment; see the matching `interface3` PDF for the API reference manual. The `expl3` modules set up a naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ε -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

The `expl3` modules are designed to be loaded on top of L^AT_EX 2 ε . With an up-to-date L^AT_EX 2 ε kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other T_EX formats, subject to restrictions on the full range of functionality.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction	1
1	Introduction to <code>expl3</code> and this document	2
1.1	Naming functions and variables	2
1.1.1	Behavior of c-type arguments when the N-type token resulting from expansion is undefined	5
1.1.2	Scratch variables	5
1.1.3	Terminological inexactitude	5
1.2	Documentation conventions	6
1.3	Formal language conventions which apply generally	7
1.4	TeX concepts not supported by L ^A T _E X3	8
II	Bootstrapping	9
2	The <code>l3bootstrap</code> module: Bootstrap code	10
2.1	Using the L ^A T _E X3 modules	10
3	The <code>l3names</code> module: Namespace for primitives	12
3.1	Setting up the L ^A T _E X3 programming language	12
III	Programming Flow	13
4	The <code>l3basics</code> module: Basic definitions	14
4.1	No operation functions	14
4.2	Grouping material	14
4.3	Control sequences and functions	15
4.3.1	Defining functions	15
4.3.2	Defining new functions using parameter text	16
4.3.3	Defining new functions using the signature	19
4.3.4	Copying control sequences	21
4.3.5	Deleting control sequences	22
4.3.6	Showing control sequences	22
4.3.7	Converting to and from control sequences	22
4.4	Analyzing control sequences	24
4.5	Using or removing tokens and arguments	25
4.5.1	Selecting tokens from delimited arguments	27
4.6	Predicates and conditionals	28
4.6.1	Tests on control sequences	29
4.6.2	Primitive conditionals	29
4.7	Starting a paragraph	31
4.8	Debugging support	31

5	The <code>l3expan</code> module: Argument expansion	33
5.1	Defining new variants	33
5.2	Methods for defining variants	34
5.3	Introducing the variants	36
5.4	Manipulating the first argument	37
5.5	Manipulating two arguments	39
5.6	Manipulating three arguments	39
5.7	Unbraced expansion	41
5.8	Preventing expansion	42
5.9	Controlled expansion	43
5.10	Internal functions	45
6	The <code>l3sort</code> module: Sorting functions	46
6.1	Controlling sorting	46
7	The <code>l3tl-analysis</code> module: Analyzing token lists	48
8	The <code>l3regex</code> module: Regular expressions in $\text{\TeX}$	49
8.1	Syntax of regular expressions	50
8.1.1	Regular expression examples	50
8.1.2	Characters in regular expressions	51
8.1.3	Characters classes	51
8.1.4	Structure: alternatives, groups, repetitions	52
8.1.5	Matching exact tokens	53
8.1.6	Miscellaneous	55
8.2	Syntax of the replacement text	55
8.3	Pre-compiling regular expressions	57
8.4	Matching	58
8.5	Submatch extraction	59
8.6	Replacement	60
8.7	Scratch regular expressions	62
8.8	Bugs, misfeatures, future work, and other possibilities	62
9	The <code>l3prg</code> module: Control structures	65
9.1	Defining a set of conditional functions	66
9.2	The boolean data type	68
9.2.1	Constant and scratch booleans	69
9.3	Boolean expressions	70
9.4	Logical loops	72
9.5	Producing multiple copies	73
9.6	Detecting $\text{\TeX}$'s mode	73
9.7	Primitive conditionals	74
9.8	Nestable recursions and mappings	74
9.8.1	Simple mappings	75
9.9	Internal programming functions	75

10 The <code>l3sys</code> module: System/runtime functions	76
10.1 The name of the job	76
10.2 Date and time	76
10.3 Engine	77
10.4 Output format	78
10.5 Platform	79
10.6 Random numbers	79
10.7 Access to the shell	79
10.8 System queries	80
10.9 Loading configuration data	81
10.9.1 Final settings	82
11 The <code>l3msg</code> module: Messages	83
11.1 Creating new messages	83
11.2 Customizable information for message modules	84
11.3 Contextual information for messages	85
11.4 Issuing messages	86
11.4.1 Messages for showing material	90
11.4.2 Expandable error messages	90
11.5 Redirecting messages	91
12 The <code>l3file</code> module: File and I/O operations	93
12.1 Input–output stream management	93
12.1.1 Reading from files	95
12.1.2 Reading from the terminal	99
12.1.3 Writing to files	99
12.1.4 Wrapping lines in output	101
12.1.5 Constant input–output streams, and variables	102
12.1.6 Primitive conditionals	102
12.2 File operations	102
12.2.1 Basic file operations	102
12.2.2 Information about files and file contents	103
12.2.3 Accessing file contents	106
13 The <code>l3luatex</code> module: Lua_{TeX}-specific functions	108
13.1 Breaking out to Lua	108
13.2 Lua interfaces	109
14 The <code>l3legacy</code> module: Interfaces to legacy concepts	111
 IV Data types	 112

15 The <code>l3tl</code> module: Token lists	113
15.1 Creating and initializing token list variables	113
15.2 Adding data to token list variables	114
15.3 Token list conditionals	115
15.3.1 Testing the first token	117
15.4 Working with token lists as a whole	118
15.4.1 Using token lists	118
15.4.2 Counting and reversing token lists	119
15.4.3 Viewing token lists	121
15.5 Manipulating items in token lists	122
15.5.1 Mapping over token lists	122
15.5.2 Head and tail of token lists	123
15.5.3 Items and ranges in token lists	125
15.5.4 Formatting token lists	127
15.5.5 Sorting token lists	127
15.6 Manipulating tokens in token lists	127
15.6.1 Replacing tokens	127
15.6.2 Reassigning category codes	129
15.7 Constant token lists	130
15.8 Scratch token lists	131
16 The <code>l3tl-build</code> module: Piecewise <code>tl</code> constructions	132
16.1 Constructing <code><tl var></code> by accumulation	132
17 The <code>l3str</code> module: Strings	134
17.1 Creating and initializing string variables	135
17.2 Adding data to string variables	135
17.3 String conditionals	136
17.4 Mapping over strings	138
17.5 Working with the content of strings	140
17.6 Modifying string variables	142
17.7 String manipulation	143
17.8 Viewing strings	144
17.9 Constant strings	145
17.10 Scratch strings	145
18 The <code>l3str-convert</code> module: String encoding conversions	146
18.1 Encoding and escaping schemes	146
18.2 Conversion functions	148
18.3 Conversion by expansion (for PDF contexts)	148
18.4 Possibilities, and things to do	148
19 The <code>l3str-format</code> package: Formatting strings of characters	150
19.1 Format specifications	150

20 The l3quark module: Quarks and scan marks	151
20.1 Quarks	151
20.2 Defining quarks	152
20.3 Quark tests	152
20.4 Recursion	153
20.4.1 An example of recursion with quarks	154
20.5 Scan marks	154
21 The l3seq module: Sequences and stacks	156
21.1 Creating and initializing sequences	156
21.2 Appending data to sequences	159
21.3 Recovering items from sequences	159
21.4 Recovering values from sequences with branching	161
21.5 Modifying sequences	162
21.6 Sequence conditionals	163
21.7 Mapping over sequences	163
21.8 Using the content of sequences directly	166
21.9 Sequences as stacks	167
21.10 Sequences as sets	168
21.11 Constant and scratch sequences	169
21.12 Viewing sequences	170
22 The l3int module: Integers	171
22.1 Integer expressions	171
22.2 Creating and initializing integers	173
22.3 Setting and incrementing integers	174
22.4 Using integers	175
22.5 Integer expression conditionals	175
22.6 Integer expression loops	177
22.7 Integer step functions	179
22.8 Formatting integers	180
22.9 Converting from other formats to integers	182
22.10 Random integers	183
22.11 Viewing integers	183
22.12 Constant integers	184
22.13 Scratch integers	184
22.14 Direct number expansion	185
22.15 Primitive conditionals	185
23 The l3flag module: Expandable flags	187
23.1 Setting up flags	187
23.2 Expandable flag commands	188

24 The <code>l3clist</code> module: Comma separated lists	190
24.1 Creating and initializing comma lists	191
24.2 Adding data to comma lists	192
24.3 Modifying comma lists	193
24.4 Comma list conditionals	194
24.5 Mapping over comma lists	194
24.6 Using the content of comma lists directly	196
24.7 Comma lists as stacks	197
24.8 Using a single item	199
24.9 Viewing comma lists	199
24.10 Constant and scratch comma lists	200
25 The <code>l3token</code> module: Token manipulation	201
25.1 Creating character tokens	202
25.2 Manipulating and interrogating character tokens	203
25.3 Generic tokens	206
25.4 Converting tokens	206
25.5 Token conditionals	207
25.6 Peeking ahead at the next token	211
25.7 Description of all possible tokens	216
26 The <code>l3prop</code> module: Property lists	219
26.1 Creating and initializing property lists	220
26.2 Adding and updating property list entries	222
26.3 Recovering values from property lists	223
26.4 Modifying property lists	224
26.5 Property list conditionals	224
26.6 Recovering values from property lists with branching	225
26.7 Mapping over property lists	226
26.8 Viewing property lists	227
26.9 Scratch property lists	228
26.10 Constants	228
27 The <code>l3skip</code> module: Dimensions and skips	229
27.1 Creating and initializing <code>dim</code> variables	229
27.2 Setting <code>dim</code> variables	230
27.3 Utilities for dimension calculations	230
27.4 Dimension expression conditionals	231
27.5 Dimension expression loops	233
27.6 Dimension step functions	234
27.7 Using <code>dim</code> expressions and variables	235
27.8 Viewing <code>dim</code> variables	237
27.9 Constant dimensions	238
27.10 Scratch dimensions	238
27.11 Inserting dimensions into the output	238
27.12 Creating and initializing <code>skip</code> variables	238
27.13 Setting <code>skip</code> variables	239
27.14 Skip expression conditionals	240
27.15 Using <code>skip</code> expressions and variables	240
27.16 Viewing <code>skip</code> variables	240

27.17	Constant skips	241
27.18	Scratch skips	241
27.19	Inserting skips into the output	241
27.20	Creating and initializing muskip variables	242
27.21	Setting muskip variables	242
27.22	Using muskip expressions and variables	243
27.23	Viewing muskip variables	243
27.24	Constant muskips	243
27.25	Scratch muskips	244
27.26	Primitive conditional	244
28	The l3keys module: Key–value interfaces	245
28.1	Creating keys	246
28.2	Sub-dividing keys	251
28.3	Choice and multiple choice keys	251
28.4	Key usage scope	254
28.5	Setting keys	254
28.5.1	Expanding the values of keys	255
28.6	Handling of unknown keys	255
28.7	Selective key setting	255
28.8	Precompiling keys	257
28.9	Utility functions for keys	258
28.10	Low-level interface for parsing key–val lists	258
29	The l3intarray module: Fast global integer arrays	262
29.1	Creating and initializing integer array variables	262
29.2	Adding data to integer arrays	263
29.3	Counting entries in integer arrays	263
29.4	Using a single entry	263
29.5	Integer array conditional	263
29.6	Viewing integer arrays	263
29.7	Implementation notes	264
30	The l3fp module: Floating points	265
30.1	Creating and initializing floating point variables	267
30.2	Setting floating point variables	267
30.3	Using floating points	268
30.4	Formatting floating points	270
30.5	Floating point conditionals	270
30.6	Floating point expression loops	271
30.7	Symbolic expressions	273
30.8	User-defined functions	275
30.9	Some useful constants, and scratch variables	276
30.10	Scratch variables	276
30.11	Floating point exceptions	277
30.12	Viewing floating points	278
30.13	Floating point expressions	278
30.13.1	Input of floating point numbers	278
30.13.2	Precedence of operators	279
30.13.3	Operations	280

30.14	Disclaimer and roadmap	287
31	The <code>l3farray</code> module: Fast global floating point arrays	290
31.1	Creating and initializing floating point array variables	290
31.2	Adding data to floating point arrays	290
31.3	Counting entries in floating point arrays	291
31.4	Using a single entry	291
31.5	Floating point array conditional	291
32	The <code>l3bitset</code> module: Bitsets	292
32.1	Creating bitsets	293
32.2	Setting and unsetting bits	294
32.3	Using bitsets	294
33	The <code>l3cctab</code> module: Category code tables	296
33.1	Creating and initializing category code tables	296
33.2	Using category code tables	297
33.3	Category code table conditionals	297
33.4	Constant and scratch category code tables	297
V	Text manipulation	299
34	The <code>l3unicode</code> module: Unicode support functions	300
35	The <code>l3text</code> module: Text processing	303
35.1	Expanding text	303
35.2	Case changing	304
35.3	Removing formatting from text	306
35.4	Control variables	306
35.5	Mapping to text	307
35.5.1	Support utilities	308
VI	Typesetting	310
36	The <code>l3box</code> module: Boxes	311
36.1	Creating and initializing boxes	311
36.2	Using boxes	312
36.3	Measuring and setting box dimensions	312
36.4	Box conditionals	313
36.5	The last box inserted	314
36.6	Constant boxes	314
36.7	Scratch boxes	314
36.8	Viewing box contents	314
36.9	Boxes and color	315
36.10	Horizontal mode boxes	315
36.11	Vertical mode boxes	316
36.12	Using boxes efficiently	318
36.13	Affine transformations	319
36.14	Framing boxes	322

36.15	Viewing part of a box	322
36.16	Primitive box conditionals	323
37	The l3coffins module: Coffin code layer	324
37.1	Controlling coffin poles	324
37.2	Creating and initializing coffins	325
37.3	Setting coffin content and poles	326
37.4	Coffin affine transformations	327
37.5	Joining and using coffins	328
37.6	Measuring coffins	328
37.7	Coffin diagnostics	329
37.8	Constants and variables	330
38	The l3color module: Color support	331
38.1	Color in boxes	331
38.2	Color models	331
38.3	Color expressions	333
38.4	Named colors	334
38.5	Selecting colors	335
38.6	Colors for fills and strokes	335
38.6.1	Coloring math mode material	335
38.7	Multiple color models	336
38.8	Exporting color specifications	336
38.9	Creating new color models	337
38.9.1	Color profiles	338
39	The l3graphics module: Graphics inclusion support	339
39.1	Graphics keys	339
39.2	Including graphics	340
39.3	Utility functions	340
39.4	Showing and logging included graphics	341
40	The l3opacity module: Opacity (transparency) support	342
40.1	Selecting opacity	342
41	The l3pdf module: Core PDF support	343
41.1	Objects	343
41.1.1	Named objects	343
41.1.2	Indexed objects	344
41.1.3	General functions	344
41.2	Version	345
41.3	Page (media) size	345
41.4	Compression	345
41.5	Destinations	346
VII	Utilities	347
42	The l3benchmark module: Benchmarking	348
42.1	Benchmark	348

VIII	Implementation	350
43	l3bootstrap implementation	351
43.1	The <code>\pdfstrcmp</code> primitive in X _Y TeX	351
43.2	Loading support Lua code	351
43.3	Engine requirements	352
43.4	The L ^A T _E X3 code environment	353
44	l3names implementation	355
45	l3kernel-functions: kernel-reserved functions	381
45.1	Internal l3debug kernel functions	381
45.2	Internal kernel functions	382
45.3	Kernel backend functions	389
46	l3basics implementation	391
46.1	Renaming some T _E X primitives (again)	391
46.2	Defining some constants	393
46.3	Defining functions	393
46.4	Selecting tokens	394
46.5	Gobbling tokens from input	397
46.6	Debugging and patching later definitions	397
46.7	Conditional processing and definitions	398
46.8	Dissecting a control sequence	404
46.9	Exist or free	406
46.10	Preliminaries for new functions	409
46.11	Defining new functions	410
46.12	Copying definitions	412
46.13	Undefining functions	413
46.14	Generating parameter text from argument count	413
46.15	Defining functions from a given number of arguments	414
46.16	Using the signature to define functions	415
46.17	Checking control sequence equality	418
46.18	Diagnostic functions	418
46.19	Decomposing a macro definition	420
46.20	Doing nothing functions	422
46.21	Breaking out of mapping functions	422
46.22	Starting a paragraph	423
47	l3expan implementation	424
47.1	General expansion	424
47.2	Hand-tuned definitions	428
47.3	Last-unbraced versions	431
47.4	Preventing expansion	433
47.5	Controlled expansion	433
47.6	Defining function variants	434
47.7	Definitions with the automated technique	444
47.8	Held-over variant generation	446

48 l3sort implementation	447
48.1 Variables	447
48.2 Finding available \toks registers	448
48.3 Protected user commands	450
48.4 Merge sort	452
48.5 Expandable sorting	455
48.6 Messages	460
49 l3tl-analysis implementation	463
49.1 Internal functions	463
49.2 Internal format	463
49.3 Variables and helper functions	464
49.4 Plan of attack	466
49.5 Disabling active characters	467
49.6 First pass	469
49.7 Second pass	474
49.8 Mapping through the analysis	477
49.9 Showing the results	478
49.10 Peeking ahead	480
49.11 Messages	487
50 l3regex implementation	489
50.1 Plan of attack	489
50.2 Helpers	490
50.2.1 Constants and variables	493
50.2.2 Testing characters	493
50.2.3 Internal auxiliaries	494
50.2.4 Character property tests	497
50.2.5 Simple character escape	499
50.3 Compiling	505
50.3.1 Variables used when compiling	506
50.3.2 Generic helpers used when compiling	507
50.3.3 Mode	508
50.3.4 Framework	510
50.3.5 Quantifiers	513
50.3.6 Raw characters	516
50.3.7 Character properties	518
50.3.8 Anchoring and simple assertions	519
50.3.9 Character classes	519
50.3.10 Groups and alternations	523
50.3.11 Catcodes and csnames	525
50.3.12 Raw token lists with \u	529
50.3.13 Other	533
50.3.14 Showing regexes	533
50.4 Building	540
50.4.1 Variables used while building	540
50.4.2 Framework	541
50.4.3 Helpers for building an NFA	544
50.4.4 Building classes	545
50.4.5 Building groups	547

50.4.6	Others	551
50.5	Matching	553
50.5.1	Variables used when matching	553
50.5.2	Matching: framework	556
50.5.3	Using states of the NFA	559
50.5.4	Actions when matching	560
50.6	Replacement	562
50.6.1	Variables and helpers used in replacement	562
50.6.2	Query and brace balance	564
50.6.3	Framework	565
50.6.4	Submatches	568
50.6.5	Csnames in replacement	570
50.6.6	Characters in replacement	571
50.6.7	An error	575
50.7	User functions	575
50.7.1	Variables and helpers for user functions	579
50.7.2	Matching	580
50.7.3	Extracting submatches	581
50.7.4	Replacement	586
50.7.5	Peeking ahead	589
50.8	Messages	595
50.9	Code for tracing	601
51	l3prg implementation	603
51.1	Primitive conditionals	603
51.2	Defining a set of conditional functions	603
51.3	The boolean data type	603
51.4	Internal auxiliaries	605
51.5	Boolean expressions	607
51.6	Logical loops	611
51.7	Producing multiple copies	613
51.8	Detecting T _E X's mode	614
51.9	Internal programming functions	615
52	l3sys implementation	617
52.1	Kernel code	617
52.1.1	Detecting the engine	617
52.1.2	Platform	620
52.1.3	Configurations	621
52.1.4	Access to the shell	623
52.2	Dynamic (every job) code	625
52.2.1	The name of the job	625
52.2.2	Time and date	626
52.2.3	Random numbers	627
52.2.4	Access to the shell	627
52.3	System queries	628
52.3.1	Held over from l3file	630
52.4	Last-minute code	630
52.4.1	Detecting the output	630
52.4.2	Configurations	631

53 l3msg implementation	632
53.1 Internal auxiliaries	632
53.2 Creating messages	632
53.3 Messages: support functions and text	634
53.4 Showing messages: low level mechanism	635
53.5 Displaying messages	637
53.6 Kernel-specific functions	646
53.7 Internal messages	647
53.8 Expandable errors	654
53.9 Message formatting	655
54 l3file implementation	657
54.1 Input operations	657
54.1.1 Variables and constants	657
54.1.2 Stream management	658
54.1.3 Reading input	661
54.2 Output operations	664
54.2.1 Variables and constants	664
54.2.2 Internal auxiliaries	665
54.3 Stream management	666
54.3.1 Deferred writing	668
54.3.2 Immediate writing	669
54.3.3 Special characters for writing	670
54.3.4 Hard-wrapping lines to a character count	670
54.4 File operations	680
54.4.1 Internal auxiliaries	681
54.5 GetIdInfo	697
54.6 Checking the version of kernel dependencies	698
54.7 Messages	700
54.8 Functions delayed from earlier modules	701
55 l3luatex implementation	702
55.1 Breaking out to Lua	702
55.2 Messages	703
55.3 Lua functions for internal use	703
55.4 Preserving iniTeX Lua data for runs	709
55.5 Parsing Unicode data files	711
56 l3legacy implementation	712

57 l3tl implementation	714
57.1 Functions	714
57.2 Constant token lists	716
57.3 Adding to token list variables	716
57.4 Internal quarks and quark-query functions	719
57.5 Reassigning token list category codes	720
57.6 Modifying token list variables	725
57.7 Token list conditionals	729
57.8 Mapping over token lists	734
57.9 Using token lists	736
57.10 Working with the contents of token lists	737
57.11 The first token from a token list	740
57.12 Token by token changes	745
57.13 Using a single item	748
57.14 Viewing token lists	751
57.15 Internal scan marks	753
57.16 Scratch token lists	753
58 l3tl-build implementation	754
59 l3str implementation	758
59.1 Internal auxiliaries	758
59.2 Creating and setting string variables	759
59.3 Modifying string variables	760
59.4 String comparisons	761
59.5 Mapping over strings	765
59.6 Accessing specific characters in a string	767
59.7 Counting characters	771
59.8 The first character in a string	773
59.9 String manipulation	774
59.10 Viewing strings	777
60 l3str-convert implementation	779
60.1 Helpers	779
60.1.1 Variables and constants	779
60.2 String conditionals	781
60.3 Conversions	782
60.3.1 Producing one byte or character	782
60.3.2 Mapping functions for conversions	783
60.3.3 Error-reporting during conversion	784
60.3.4 Framework for conversions	785
60.3.5 Byte unescape and escape	789
60.3.6 Native strings	790
60.3.7 clist	791
60.3.8 8-bit encodings	791
60.4 Messages	794
60.5 Escaping definitions	795
60.5.1 Unescape methods	796
60.5.2 Escape methods	800
60.6 Encoding definitions	802

60.6.1	UTF-8 support	802
60.6.2	UTF-16 support	807
60.6.3	UTF-32 support	812
60.7	PDF names and strings by expansion	815
60.7.1	ISO 8859 support	816
61	l3str-format implementation	833
61.1	Helpers	833
61.2	Parsing a format specification	834
61.3	Alignment	835
61.4	Formatting token lists	837
61.5	Formatting sequences	838
61.6	Formatting integers	839
61.7	Formatting floating points	841
61.8	Messages	845
62	l3quark implementation	846
62.1	Quarks	846
62.2	Scan marks	854
63	l3seq implementation	856
63.1	Allocation and initialization	857
63.2	Appending data to either end	861
63.3	Modifying sequences	862
63.4	Sequence conditionals	866
63.5	Recovering data from sequences	868
63.6	Mapping over sequences	872
63.7	Using sequences	877
63.8	Sequence stacks	878
63.9	Viewing sequences	878
63.10	Scratch sequences	879
64	l3int implementation	880
64.1	Integer expressions	881
64.2	Creating and initializing integers	883
64.3	Setting and incrementing integers	885
64.4	Using integers	887
64.5	Integer expression conditionals	887
64.6	Integer expression loops	891
64.7	Integer step functions	892
64.8	Formatting integers	894
64.9	Converting from other formats to integers	900
64.10	Viewing integer	903
64.11	Random integers	903
64.12	Constant integers	904
64.13	Scratch integers	904
64.14	Integers for earlier modules	904

65 l3flag implementation	905
65.1 Protected flag commands	905
65.2 Expandable flag commands	906
65.3 Old n-type flag commands	907
66 l3clist implementation	909
66.1 Removing spaces around items	910
66.2 Allocation and initialization	911
66.3 Adding data to comma lists	913
66.4 Comma lists as stacks	914
66.5 Modifying comma lists	916
66.6 Comma list conditionals	919
66.7 Mapping over comma lists	920
66.8 Using comma lists	924
66.9 Using a single item	926
66.10 Viewing comma lists	928
66.11 Scratch comma lists	929
67 l3token implementation	930
67.1 Internal auxiliaries	930
67.2 Manipulating and interrogating character tokens	930
67.3 Creating character tokens	933
67.4 Generic tokens	936
67.5 Token conditionals	938
67.6 Peeking ahead at the next token	949
68 l3prop implementation	956
68.1 Internal auxiliaries	957
68.2 Structure of a property list	958
68.3 Allocation and initialization	960
68.4 Accessing data in property lists	967
68.5 Removing data from property lists	970
68.6 Adding data to property lists	973
68.7 Property list conditionals	975
68.8 Mapping over property lists	977
68.9 Uses of mapping over property lists	979
68.10 Viewing property lists	980
69 l3skip implementation	984
69.1 Length primitives renamed	984
69.2 Internal auxiliaries	984
69.3 Creating and initializing dim variables	984
69.4 Setting dim variables	985
69.5 Utilities for dimension calculations	986
69.6 Dimension expression conditionals	987
69.7 Dimension expression loops	990
69.8 Dimension step functions	991
69.9 Using dim expressions and variables	992
69.10 Conversion of dim to other units	994
69.11 Viewing dim variables	998

69.12	Constant dimensions	999
69.13	Scratch dimensions	999
69.14	Inserting dimensions into the output	999
69.15	Creating and initializing skip variables	999
69.16	Setting skip variables	1001
69.17	Skip expression conditionals	1001
69.18	Using skip expressions and variables	1002
69.19	Inserting skips into the output	1002
69.20	Viewing skip variables	1003
69.21	Constant skips	1003
69.22	Scratch skips	1003
69.23	Creating and initializing muskip variables	1003
69.24	Setting muskip variables	1004
69.25	Using muskip expressions and variables	1005
69.26	Viewing muskip variables	1005
69.27	Constant muskips	1006
69.28	Scratch muskips	1006
70	l3keys implementation	1007
70.1	Low-level interface	1007
70.2	Constants and variables	1014
70.2.1	Internal auxiliaries	1017
70.3	The key defining mechanism	1018
70.4	Turning properties into actions	1021
70.5	Creating key properties	1028
70.6	Setting keys	1034
70.7	Utilities	1044
70.8	Messages	1047
71	l3intarray implementation	1049
71.1	Lua implementation	1049
71.1.1	Allocating arrays	1049
71.1.2	Array items	1052
71.1.3	Working with contents of integer arrays	1054
71.2	Font dimension based implementation	1055
71.2.1	Allocating arrays	1056
71.2.2	Array items	1057
71.2.3	Working with contents of integer arrays	1059
71.3	Common parts	1061
72	l3fp implementation	1062

73 l3fp-aux implementation	1063
73.1 Access to primitives	1063
73.2 Internal representation	1063
73.3 Using arguments and \@@_sep:s	1064
73.4 Constants, and structure of floating points	1065
73.5 Overflow, underflow, and exact zero	1068
73.6 Expanding after a floating point number	1068
73.7 Other floating point types	1069
73.8 Packing digits	1072
73.9 Decimate (dividing by a power of 10)	1075
73.10 Functions for use within primitive conditional branches	1077
73.11 Integer floating points	1078
73.12 Small integer floating points	1079
73.13 Fast string comparison	1080
73.14 Name of a function from its l3fp-parse name	1080
73.15 Messages	1080
74 l3fp-traps implementation	1081
74.1 Flags	1081
74.2 Traps	1081
74.3 Errors	1085
74.4 Messages	1086
75 l3fp-round implementation	1087
75.1 Rounding tools	1087
75.2 The round function	1091
76 l3fp-parse implementation	1096
76.1 Work plan	1096
76.1.1 Storing results	1097
76.1.2 Precedence and infix operators	1098
76.1.3 Prefix operators, parentheses, and functions	1101
76.1.4 Numbers and reading tokens one by one	1102
76.2 Main auxiliary functions	1104
76.3 Helpers	1105
76.4 Parsing one number	1106
76.4.1 Numbers: trimming leading zeros	1112
76.4.2 Number: small significand	1113
76.4.3 Number: large significand	1115
76.4.4 Number: beyond 16 digits, rounding	1117
76.4.5 Number: finding the exponent	1120
76.5 Constants, functions and prefix operators	1123
76.5.1 Prefix operators	1123
76.5.2 Constants	1126
76.5.3 Functions	1127
76.6 Main functions	1128
76.7 Infix operators	1130
76.7.1 Closing parentheses and commas	1132
76.7.2 Usual infix operators	1133
76.7.3 Juxtaposition	1134

76.7.4	Multi-character cases	1134
76.7.5	Ternary operator	1135
76.7.6	Comparisons	1135
76.8	Tools for functions	1138
76.9	Messages	1140
77	l3fp-assign implementation	1141
77.1	Assigning values	1141
77.2	Updating values	1142
77.3	Showing values	1142
77.4	Some useful constants and scratch variables	1144
78	l3fp-logic implementation	1146
78.1	Syntax of internal functions	1146
78.2	Tests	1146
78.3	Comparison	1147
78.4	Floating point expression loops	1150
78.5	Extrema	1154
78.6	Boolean operations	1155
78.7	Ternary operator	1156
79	l3fp-basics implementation	1158
79.1	Addition and subtraction	1158
79.1.1	Sign, exponent, and special numbers	1159
79.1.2	Absolute addition	1161
79.1.3	Absolute subtraction	1163
79.2	Multiplication	1168
79.2.1	Signs, and special numbers	1168
79.2.2	Absolute multiplication	1169
79.3	Division	1171
79.3.1	Signs, and special numbers	1171
79.3.2	Work plan	1172
79.3.3	Implementing the significand division	1175
79.4	Square root	1180
79.5	About the sign and exponent	1187
79.6	Operations on tuples	1188
80	l3fp-extended implementation	1190
80.1	Description of fixed point numbers	1190
80.2	Helpers for numbers with extended precision	1191
80.3	Multiplying a fixed point number by a short one	1192
80.4	Dividing a fixed point number by a small integer	1192
80.5	Adding and subtracting fixed points	1193
80.6	Multiplying fixed points	1194
80.7	Combining product and sum of fixed points	1195
80.8	Extended-precision floating point numbers	1198
80.9	Dividing extended-precision numbers	1200
80.10	Inverse square root of extended precision numbers	1204
80.11	Converting from fixed point to floating point	1206

81 l3fp-expo implementation	1208
81.1 Logarithm	1208
81.1.1 Work plan	1208
81.1.2 Some constants	1209
81.1.3 Sign, exponent, and special numbers	1209
81.1.4 Absolute ln	1210
81.2 Exponential	1217
81.2.1 Sign, exponent, and special numbers	1217
81.3 Power	1221
81.4 Factorial	1228
82 l3fp-trig implementation	1231
82.1 Direct trigonometric functions	1232
82.1.1 Filtering special cases	1232
82.1.2 Distinguishing small and large arguments	1235
82.1.3 Small arguments	1236
82.1.4 Argument reduction in degrees	1236
82.1.5 Argument reduction in radians	1237
82.1.6 Computing the power series	1245
82.2 Inverse trigonometric functions	1248
82.2.1 Arctangent and arccotangent	1249
82.2.2 Arcsine and arccosine	1254
82.2.3 Arccosecant and arcsecant	1256
83 l3fp-convert implementation	1258
83.1 Dealing with tuples	1258
83.2 Trimming trailing zeros	1258
83.3 Scientific notation	1259
83.4 Decimal representation	1260
83.5 Token list representation	1262
83.6 Formatting	1263
83.7 Convert to dimension or integer	1264
83.8 Convert from a dimension	1264
83.9 Use and eval	1265
83.10 Convert an array of floating points to a comma list	1266
84 l3fp-random implementation	1268
84.1 Engine support	1268
84.2 Random floating point	1271
84.3 Random integer	1272
85 l3fp-types implementation	1277
85.1 Support for types	1277
85.2 Dispatch according to the type	1277

86 l3fp-symbolic implementation	1280
86.1 Misc	1280
86.2 Building blocks for expressions	1280
86.3 Expanding after a symbolic expression	1282
86.4 Applying infix operators to expressions	1283
86.5 Applying prefix functions to expressions	1283
86.6 Conversions	1284
86.7 Identifiers	1285
86.8 Declaring variables and assigning values	1286
86.9 Messages	1289
86.10 Road-map	1290
87 l3fp-functions implementation	1291
87.1 Declaring functions	1291
87.2 Defining functions by their expression	1292
88 l3farray implementation	1295
88.1 Allocating arrays	1295
88.2 Array items	1296
89 l3bitset implementation	1300
89.1 Messages	1304
90 l3cctab implementation	1306
90.1 Variables	1306
90.2 Allocating category code tables	1307
90.3 Saving category code tables	1308
90.4 Using category code tables	1309
90.5 Category code table conditionals	1314
90.6 Constant category code tables	1316
90.7 Messages	1317
91 l3unicode implementation	1319
91.1 User functions	1319
91.2 Data loader	1323
92 l3text implementation	1337
92.1 Internal auxiliaries	1337
92.2 Utilities	1338
92.3 Codepoint utilities	1341
92.4 Configuration variables	1343
92.5 Expansion to formatted text	1345
93 l3text-case implementation	1354
93.1 Case changing	1354

94 l3text-map implementation	1388
94.1 Mapping to text	1388
94.1.1 Common code	1388
94.2 Grapheme mapping	1394
94.3 Word break mapping	1396
94.4 Inline mappings	1400
95 l3text-purify implementation	1401
95.1 Purifying text	1401
95.2 Accent and letter-like data for purifying text	1406
96 l3text-utils implementation	1413
96.1 Parsing BCP 47 strings	1413
97 l3box implementation	1421
97.1 Support code	1421
97.2 Creating and initializing boxes	1421
97.3 Measuring and setting box dimensions	1422
97.4 Using boxes	1423
97.5 Box conditionals	1423
97.6 The last box inserted	1424
97.7 Constant boxes	1424
97.8 Scratch boxes	1424
97.9 Viewing box contents	1425
97.10 Horizontal mode boxes	1426
97.11 Vertical mode boxes	1428
97.12 Affine transformations	1433
97.13 Adding frames to boxes	1442
97.14 Viewing part of a box	1444
98 l3coffins implementation	1447
98.1 Coffins: data structures and general variables	1447
98.2 Basic coffin functions	1448
98.3 Measuring coffins	1454
98.4 Coffins: handle and pole management	1454
98.5 Coffins: calculation of pole intersections	1457
98.6 Affine transformations	1460
98.7 Aligning and typesetting of coffins	1468
98.8 Coffin diagnostics	1473
98.9 Messages	1479

99 l3color implementation	1480
99.1 Basics	1480
99.2 Predefined color names	1481
99.3 Setup	1482
99.4 Utility functions	1482
99.5 Model conversion	1483
99.6 Color expressions	1484
99.7 Selecting colors (and color models)	1495
99.8 Math color	1497
99.9 Fill and stroke color	1500
99.10 Defining named colors	1500
99.11 Exporting colors	1503
99.12 Additional color models	1505
99.13 Applying profiles	1519
99.14 Diagnostics	1520
99.15 Messages	1521
100 l3graphics implementation	1524
100.1 Graphics keys	1524
100.2 Obtaining bounding box data	1525
100.3 Utility functions	1531
100.4 Messages	1533
101 l3opacity implementation	1534
102 l3pdf implementation	1536
102.1 Compression	1536
102.2 Objects	1537
102.3 Version	1541
102.4 Page size	1542
102.5 Destinations	1543
102.6 PDF Page size (media box)	1543
103 l3benchmark implementation	1545
103.1 Benchmarking code	1545
103.1.1 Raw measurement	1545
103.1.2 Main benchmarking	1546
103.1.3 Display	1549
103.2 Benchmark tic toc	1550

1043deprecation implementation	1552
104.1 Patching definitions to deprecate	1552
104.2 Deprecated l3basics functions	1554
104.3 Deprecated l3file functions	1554
104.4 Deprecated l3keys functions	1554
104.5 Deprecated l3msg functions	1555
104.6 Deprecated l3pdf functions	1555
104.7 Deprecated l3prg functions	1556
104.8 Deprecated l3regex functions	1556
104.9 Deprecated l3str functions	1556
104.10Deprecated l3seq functions	1557
104.11Deprecated l3sys functions	1558
104.12Deprecated l3text functions	1558
104.13Deprecated l3tl functions	1558
104.14Deprecated l3token functions	1559
104.15Deprecated l3prop functions	1561
1053debug implementation	1562
Index	1587

Part I
Introduction

Chapter 1

Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the expl3 language. A general guide to the L^AT_EX3 programming language is found in [expl3.pdf](#).

1.1 Naming functions and variables

L^AT_EX3 does not use @ as a “letter” for defining internal macros. Instead, the symbols _ and : are used in internal macro names to provide structure. The name of each *function* is divided into logical units using _, while : separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all expl3 function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

- N and n** These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.
- c** This means *cname*, and indicates that the argument will be turned into a `cname` before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`. All macros that appear in the argument are expanded. An internal error will occur if the result of expansion inside a `c`-type argument is not a series of character tokens.
- V and v** These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying T_EX structure containing the data. A `V` argument will be a single token (similar to `N`), for example

`\foo:V \MyVariable`; on the other hand, using `v` a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

- o This means *expansion once*. In general, the `V` and `v` specifiers are favored over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.
- x The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The TeX `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are *not* expandable.
- e The `e` specifier is in many respects identical to `x`, but uses `\expanded` primitive. Parameter character (usually `#`) in the argument need not be doubled. Functions which feature an `e`-type argument may be expandable.
- f The `f` specifier stands for *full expansion*, and in contrast to `x` stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *space token*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` is considered.

- T and F** For logic tests, there are the branch specifiers `T` (*true*) and `F` (*false*). Both specifiers treat the input in the same way as `n` (no change), but make the logic much easier to see.
- p The letter `p` indicates TeX *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w Finally, there is the `w` specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- D The `D` stands for **Do not use**. All of the TeX primitives are initially `\let` to a `D` name, and some are then given a second name. These functions have no standardized syntax, they are engine dependent and their name can change without warning, thus their use is *strongly discouraged* in package code: programmers should instead use the interfaces documented in [interface3.pdf](#).

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

- c Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

bitset a set of bits (a string made up of a series of 0 and 1 tokens that are accessed by position).

clist Comma separated list.

dim “Rigid” lengths.

fp Floating-point values;

int Integer-valued count register.

muskip “Rubber” lengths for use in mathematics.

skip “Rubber” lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying **V**-type or **v**-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

flag Non-negative integer that can be incremented expandably.

fparray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

seq “Sequence”: a data type used to implement lists (with access at both ends) and stacks.

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called `\l_tmpa_int`, `\l_tmpl_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

1.1.1 Behavior of c-type arguments when the N-type token resulting from expansion is undefined

When c-type expansion is applied, it will produce an N-type token to be consumed by the underlying function. If the result of this process is a token which is undefined, T_EX’s behavior is to make it equal to `\scan_stop: (\relax)`.

This will likely lead to low-level errors if it occurs in contexts where `expl3` expects a “variable”, e.g. a `prop`, `seq`, etc. Therefore, the programmer should ensure that c-type expansion is only applied when the resulting N-type token will definitely exist, i.e., when it is either defined prior to the application of the c-type expansion or will be by the underlying N-type function.

1.1.2 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form `\<scope>_tmpa_<type>/\<scope>_tmpb_<type>`. These are never used by the core code. The nature of T_EX grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

There are two more special types of constants:

q Quark constants.

s Scan mark constants.

Similarly, each quark or scan mark name starts with the module name, but doesn’t end with a variable type, because the type is already marked by the prefix **q** or **s**. Some general quarks and scan marks provided by L^AT_EX3 don’t start with a module name, for example `\s_stop`. See documentation of quarks and scan marks in Chapter VIII for more info.

1.1.3 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, T_EX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.² On the other hand, some “variables” are actually registers that must be initialized and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in T_EX’s stomach” (if you are familiar with the T_EXbook parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

²T_EXnically, functions with no arguments are `\long` while token list variables are not.

1.2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

<code>\ExplSyntaxOn</code> <code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn ... \ExplSyntaxOff</code> The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

<code>\seq_new:N</code> <code>\seq_new:c</code>	<code>\seq_new:N <seq var></code> When a number of variants are described, the arguments are usually illustrated only for the base function. Here, <code><seq var></code> indicates that <code>\seq_new:N</code> expects a sequence variable. From the argument specifier, <code>\seq_new:c</code> also expects a sequence variable, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type or `e`-type argument (in plain $\TeX$ terms, inside an `\edef` or `\expanded`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

<code>\cs_to_str:N</code> ☆	<code>\cs_to_str:N <cs></code> As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a <code><cs></code> , shorthand for a <code><control sequence></code> .

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

<code>\seq_map_function:NN</code> ☆	<code>\seq_map_function:NN <seq var> <function></code>

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

`\sys_if_engine_xetex:TF *` `\sys_if_engine_xetex:TF {<true code>} {<false code>}`

The underlining and italic of TF indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the TF variant, and so both *<true code>* and *<false code>* will be shown. The two variant forms T and F take only *<true code>* and *<false code>*, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

`\l_tmpa_tl` A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in $\text{\LaTeX} 2_{\epsilon}$ or plain $\text{\TeX}$. In these cases, the text will include an extra “ **$\text{\TeX}$ hackers note**” section:

`\token_to_str:N *` `\token_to_str:N <token>`

The normal description text.

$\text{\TeX}$ hackers note: Detail for the experienced $\text{\TeX}$ or $\text{\LaTeX} 2_{\epsilon}$ programmer. In this case, it would point out that this function is the $\text{\TeX}$ primitive `\string`.

Addition dates For functions added to `expl3` after 2020-02-02 (the point at which it was integrated into the $\text{\LaTeX}$ kernel), the date of addition is included in the documentation as “New”.

Changes to behavior Where the documented behavior of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behavior as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behavior.

1.3 Formal language conventions which apply generally

As this is a formal reference guide for $\text{\LaTeX} 3$ programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarized here.

For tests which have a TF argument specification, the test is evaluated to give a logically TRUE or FALSE result. Depending on this result, either the *<true code>* or the *<false code>* will be left in the input stream. In the case where the test is expandable,

and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

1.4 $\text{\TeX}$ concepts not supported by $\text{\LaTeX}3$

The $\text{\TeX}$ concept of an “`\outer`” macro is *not supported* at all by $\text{\LaTeX}3$. As such, the functions provided here may break when used on top of $\text{\LaTeX}2_\epsilon$ if `\outer` tokens are used in the arguments.

Part II

Bootstrapping

Chapter 2

The l3bootstrap module

Bootstrap code

2.1 Using the L^AT_EX3 modules

The modules documented in `interface3` (and this file) are designed to be used on top of L^AT_EX 2_ε and are already pre-loaded since L^AT_EX 2_ε 2020-02-02. To support older formats, the `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions are still available to load them all as one.

As the modules use a coding syntax different from standard L^AT_EX 2_ε it provides a few functions for setting it up.

<code>\ExplSyntaxOn</code>	<code>\ExplSyntaxOn <code> \ExplSyntaxOff</code>
<code>\ExplSyntaxOff</code>	

The `\ExplSyntaxOn` function switches to a category code régime in which spaces and new lines are ignored, and in which the colon (`:`) and underscore (`_`) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code régime.

T_EXhackers note: Spaces introduced by `~` behave much in the same way as normal space characters in the standard category code régime: they are ignored after a control word or at the start of a line, and multiple consecutive `~` are equivalent to a single one. However, `~` is *not* ignored at the end of a line.

<code>\ProvidesExplPackage</code>	<code>\ProvidesExplPackage {<package>} {<date>} {<version>} {<description>}</code>
<code>\ProvidesExplClass</code>	
<code>\ProvidesExplFile</code>	

Updated: 2023-08-03

These functions act broadly in the same way as the corresponding L^AT_EX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_ε provides in turning on `\makeatletter` within package and class code.) The `<date>` should be given in the format `<year>/<month>/<day>` or in the ISO date format `<year>-<month>-<day>`. If the `<version>` is given then a leading `v` is optional: if given as a “pure” version string, a `v` will be prepended.

`\GetIdInfo` `\GetIdInfo $Id: <SVN info field> $ {(description)}`

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual $\text{\LaTeX} 2_{\epsilon}$ category codes and the $\text{\LaTeX} 3$ category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}  
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

Chapter 3

The l3names module

Namespace for primitives

3.1 Setting up the L^AT_EX3 programming language

This module is at the core of the L^AT_EX3 programming language. It performs the following tasks:

- defines new names for all T_EX primitives;
- emulate required primitives not provided by default in LuaT_EX;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_YT_EX, LuaT_EX, pT_EX and upT_EX should be consulted for details of the primitives. These are named `\tex_⟨name⟩:D`, typically based on the primitive’s `⟨name⟩` in pdfT_EX and omitting a leading `pdf` when the primitive is not related to pdf output.

Part III

Programming Flow

Chapter 4

The l3basics module

Basic definitions

As the name suggests, this module holds some basic definitions which are needed by most or all other modules in this set.

Here we describe those functions that are used all over the place. By that, we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

4.1 No operation functions

<code>\prg_do_nothing:</code>	★	<code>\prg_do_nothing:</code>

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

<code>\scan_stop:</code>	<code>\scan_stop:</code>

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

4.2 Grouping material

<code>\group_begin:</code>	<code>\group_begin:</code>
<code>\group_end:</code>	<code>\group_end:</code>

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

T_EXhackers note: These are the T_EX primitives `\begingroup` and `\endgroup`.

<code>\group_insert_after:N</code>	<code>\group_insert_after:N</code> $\langle token \rangle$

Adds $\langle token \rangle$ to the list of $\langle tokens \rangle$ to be inserted when the current group level ends. The list of $\langle tokens \rangle$ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one $\langle token \rangle$ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

TeXhackers note: This is the TeX primitive `\aftergroup`.

<code>\group_show_list:</code>	<code>\group_show_list:</code>
<code>\group_log_list:</code>	<code>\group_log_list:</code>

New: 2021-05-11

Display (to the terminal or log file) a list of the groups that are currently opened. This is intended for tracking down problems.

TeXhackers note: This is a wrapper around the ϵ -TeX primitive `\showgroups`.

4.3 Control sequences and functions

As TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (`#1`, `#2`, etc.) is replaced the appropriate arguments absorbed by the function. In the following, $\langle code \rangle$ is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an **e**-type or **x**-type expansion. In contrast, “protected” functions are not expanded within **e** and **x** expansions.

4.3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

new Create a new function with the **new** scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the **set** scope, such as `\cs_set:Npn`. The definition is restricted to the current TeX group and does not result in an error if the function is already defined.

gset Create a new function with the **gset** scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the **nopar** restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the **protected** restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an **e**-type or **x**-type expansion.

Finally, the functions in Subsections 4.3.2 and 4.3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and **n** No manipulation.

T and **F** Functionally equivalent to **n** (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 9.1).

p and **w** These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 5.2.

4.3.2 Defining new functions using parameter text

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
<code>\cs_new:cpn</code>	
<code>\cs_new:Npe</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new:cpe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The
<code>\cs_new:Npx</code>	definition is global and an error results if the <code><function></code> is already defined.
<code>\cs_new:cpx</code>	

Updated: 2023-09-27

<code>\cs_new_nopar:Npn</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_nopar:cpn</code>	
<code>\cs_new_nopar:Npe</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_nopar:cpe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When
<code>\cs_new_nopar:Npx</code>	the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The
<code>\cs_new_nopar:cpx</code>	definition is global and an error results if the <code><function></code> is already defined.

Updated: 2023-09-27

<code>\cs_new_protected:Npn</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected:cpn</code>	
<code>\cs_new_protected:Npe</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_protected:cpe</code>	<code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:Npx</code>	<code><function></code> will not expand within an e -type or x -type argument. The definition is
<code>\cs_new_protected:cpx</code>	global and an error results if the <code><function></code> is already defined.

Updated: 2023-09-27

<code>\cs_new_protected_nopar:Npn</code>	<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected_nopar:cpn</code>	
<code>\cs_new_protected_nopar:Npe</code>	
<code>\cs_new_protected_nopar:cpe</code>	
<code>\cs_new_protected_nopar:Npx</code>	
<code>\cs_new_protected_nopar:cpx</code>	

Updated: 2023-09-27

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an e-type or x-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_set:Npn</code>	<code>\cs_set:Npn <function> <parameters> {<code>}</code>
<code>\cs_set:cpn</code>	
<code>\cs_set:Npe</code>	
<code>\cs_set:cpe</code>	
<code>\cs_set:Npx</code>	
<code>\cs_set:cpx</code>	

Updated: 2023-09-27

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

<code>\cs_set_nopar:Npn</code>	<code>\cs_set_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_nopar:cpn</code>	
<code>\cs_set_nopar:Npe</code>	
<code>\cs_set_nopar:cpe</code>	
<code>\cs_set_nopar:Npx</code>	
<code>\cs_set_nopar:cpx</code>	

Updated: 2023-09-27

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	
<code>\cs_set_protected:Npe</code>	
<code>\cs_set_protected:cpe</code>	
<code>\cs_set_protected:Npx</code>	
<code>\cs_set_protected:cpx</code>	

Updated: 2023-09-27

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an e-type or x-type argument.

<code>\cs_set_protected_nopar:Npn</code>	<code>\cs_set_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected_nopar:cpn</code>	
<code>\cs_set_protected_nopar:Npe</code>	
<code>\cs_set_protected_nopar:cpe</code>	
<code>\cs_set_protected_nopar:Npx</code>	
<code>\cs_set_protected_nopar:cpx</code>	

Updated: 2023-09-27

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an e-type or x-type argument.

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	
<code>\cs_gset:Npe</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current TeX group level: the assignment is global.
<code>\cs_gset:cpe</code>	
<code>\cs_gset:Npx</code>	
<code>\cs_gset:cpx</code>	

Updated: 2023-09-27

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:Npe</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current TeX group level: the assignment is global.
<code>\cs_gset_nopar:cpe</code>	
<code>\cs_gset_nopar:Npx</code>	
<code>\cs_gset_nopar:cpx</code>	

Updated: 2023-09-27

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	
<code>\cs_gset_protected:Npe</code>	Globally sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is <i>not</i> restricted to the current TeX group level: the assignment is global. The <code><function></code> will not expand within an e-type or x-type argument.
<code>\cs_gset_protected:cpe</code>	
<code>\cs_gset_protected:Npx</code>	
<code>\cs_gset_protected:cpx</code>	

Updated: 2023-09-27

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected_nopar:cpn</code>	
<code>\cs_gset_protected_nopar:Npe</code>	
<code>\cs_gset_protected_nopar:cpe</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpx</code>	

Updated: 2023-09-27

Globally sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is *not* restricted to the current T_EX group level: the assignment is global. The `<function>` will not expand within an e-type or x-type argument.

4.3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {<code>}</code>
<code>\cs_new:(cn Ne ce)</code>	

Updated: 2023-09-27

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the number of `<parameters>` is detected automatically from the function signature. These `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_new_nopar:Nn</code>	<code>\cs_new_nopar:Nn <function> {<code>}</code>
<code>\cs_new_nopar:(cn Ne ce)</code>	

Updated: 2023-09-27

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the number of `<parameters>` is detected automatically from the function signature. These `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_new_protected:Nn</code>	<code>\cs_new_protected:Nn <function> {<code>}</code>
<code>\cs_new_protected:(cn Ne ce)</code>	

Updated: 2023-09-27

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the number of `<parameters>` is detected automatically from the function signature. These `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. The `<function>` will not expand within an e-type or x-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_new_protected_nopar:Nn</code>	<code>\cs_new_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_new_protected_nopar:(cn Ne ce)</code>	

Updated: 2023-09-27

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the number of `<parameters>` is detected automatically from the function signature. These `<parameters>` (`#1`, `#2`, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an e-type or x-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_set:Nn</code>	<code>\cs_set:Nn <function> {<code>}</code>
<code>\cs_set:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is restricted to the current TeX group level.
Updated: 2023-09-27	

<code>\cs_set_nopar:Nn</code>	<code>\cs_set_nopar:Nn <function> {<code>}</code>
<code>\cs_set_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is restricted to the current TeX group level.
Updated: 2023-09-27	

<code>\cs_set_protected:Nn</code>	<code>\cs_set_protected:Nn <function> {<code>}</code>
<code>\cs_set_protected:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The <code><function></code> will not expand within an e-type or x-type argument. The assignment of a meaning to the <code><function></code> is restricted to the current TeX group level.
Updated: 2023-09-27	

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The <code><function></code> will not expand within an e-type or x-type argument. The assignment of a meaning to the <code><function></code> is restricted to the current TeX group level.
Updated: 2023-09-27	

<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <code><function></code> is global.
Updated: 2023-09-27	

<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Ne ce)</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the number of <code><parameters></code> is detected automatically from the function signature. These <code><parameters></code> (<code>#1</code> , <code>#2</code> , etc.) will be replaced by those absorbed by the function. When the <code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <code><function></code> is global.
Updated: 2023-09-27	

<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Ne ce)</code>	

Updated: 2023-09-27

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an $\mathbf{e}$ -type or $\mathbf{x}$ -type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Ne ce)</code>	

Updated: 2023-09-27

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash\mathbf{par}$ tokens. The $\langle function \rangle$ will not expand within an $\mathbf{e}$ -type or $\mathbf{x}$ -type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator> {<number>}</code>
<code>\cs_generate_from_arg_count:(NNno cNnn Ncnn)</code>	<code>{<code>}</code>

Uses the $\langle creator \rangle$ function (which should have signature $\mathbf{Npn}$, for example $\backslash\mathbf{cs\_new:Npn}$) to define a $\langle function \rangle$ which takes $\langle number \rangle$ arguments and has $\langle code \rangle$ as replacement text. The $\langle number \rangle$ of arguments is an integer expression, evaluated as detailed for $\backslash\mathbf{int\_eval:n}$.

4.3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

<code>\cs_new_eq:NN</code>	<code>\cs_new_eq:NN <cs₁> <cs₂></code>
<code>\cs_new_eq:(Nc cN cc)</code>	<code>\cs_new_eq:NN <cs₁> <token></code>

Globally creates $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. The second control sequence may subsequently be altered without affecting the copy.

<code>\cs_set_eq:NN</code>	<code>\cs_set_eq:NN <cs₁> <cs₂></code>
<code>\cs_set_eq:(Nc cN cc)</code>	<code>\cs_set_eq:NN <cs₁> <token></code>

Sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is restricted to the current $\mathbf{T_{E}X}$ group level.

<code>\cs_gset_eq:NN</code>	<code>\cs_gset_eq:NN <cs₁> <cs₂></code>
<code>\cs_gset_eq:(Nc cN cc)</code>	<code>\cs_gset_eq:NN <cs₁> <token></code>

Globally sets $\langle \textit{control sequence}_1 \rangle$ to have the same meaning as $\langle \textit{control sequence}_2 \rangle$ (or $\langle \textit{token} \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle \textit{control sequence}_1 \rangle$ is *not* restricted to the current T_EX group level: the assignment is global.

4.3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

<code>\cs_undefine:N</code>	<code>\cs_undefine:N <control sequence></code>
<code>\cs_undefine:c</code>	Sets $\langle \textit{control sequence} \rangle$ to be globally undefined.

4.3.6 Showing control sequences

<code>\cs_meaning:N *</code>	<code>\cs_meaning:N <control sequence></code>
<code>\cs_meaning:c *</code>	This function expands to the <i>meaning</i> of the $\langle \textit{control sequence} \rangle$ control sequence. For a macro, this includes the $\langle \textit{replacement text} \rangle$.

T_EXhackers note: This is the T_EX primitive `\meaning`. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

<code>\cs_show:N</code>	<code>\cs_show:N <control sequence></code>
<code>\cs_show:c</code>	Displays the definition of the $\langle \textit{control sequence} \rangle$ on the terminal.

T_EXhackers note: This is similar to the T_EX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\cs_log:N</code>	<code>\cs_log:N <control sequence></code>
<code>\cs_log:c</code>	Writes the definition of the $\langle \textit{control sequence} \rangle$ in the log file. See also <code>\cs_show:N</code> which displays the result in the terminal.

4.3.7 Converting to and from control sequences

<code>\use:c *</code>	<code>\use:c {\langle control sequence name \rangle}</code>

Expands the $\langle \textit{control sequence name} \rangle$ until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the $\langle \textit{control sequence name} \rangle$ must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

As an example of the `\use:c` function, both

`\use:c { a b c }`

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

`\abc`

after two expansions of `\use:c`.

<code>\cs_if_exist_use:N</code>	★ <code>\cs_if_exist_use:N</code> $\langle$ <i>control sequence</i> $\rangle$
<code>\cs_if_exist_use:c</code>	★ <code>\cs_if_exist_use:NTF</code> $\langle$ <i>control sequence</i> $\rangle$ $\{\langle$ <i>true code</i> $\rangle\}$ $\{\langle$ <i>false code</i> $\rangle\}$
<code>\cs_if_exist_use:NTF</code>	★ Tests whether the $\langle$ <i>control sequence</i> $\rangle$ is currently defined according to the conditional
<code>\cs_if_exist_use:cTF</code>	★ <code>\cs_if_exist:NTF</code> (whether as a function or another control sequence type), and if it is inserts the $\langle$ <i>control sequence</i> $\rangle$ into the input stream followed by the $\langle$ <i>true code</i> $\rangle$. Otherwise the $\langle$ <i>false code</i> $\rangle$ is used.

<code>\cs:w</code>	★ <code>\cs:w</code> $\langle$ <i>control sequence name</i> $\rangle$ <code>\cs_end:</code>
<code>\cs_end:</code>	★ Converts the given $\langle$ <i>control sequence name</i> $\rangle$ into a single control sequence token. This process requires one expansion. The content for $\langle$ <i>control sequence name</i> $\rangle$ may be literal material or from other expandable functions. The $\langle$ <i>control sequence name</i> $\rangle$ must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TeXhackers note: These are the TeX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

`\cs:w a b c \cs_end:`

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

`\abc`

after one expansion of `\cs:w`.

<code>\cs_to_str:N</code>	★ <code>\cs_to_str:N</code> $\langle$ <i>control sequence</i> $\rangle$

Converts the given $\langle$ *control sequence* $\rangle$ into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an e-type or x-type expansion, or two o-type expansions are required to convert the $\langle$ *control sequence* $\rangle$ to a sequence of characters in the input stream. In most cases, an f-expansion is correct as well, but this loses a space at the start of the result.

4.4 Analyzing control sequences

`\cs_split_function:N` ★ `\cs_split_function:N` $\langle function \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (i.e., the part before the colon) and the $\langle signature \rangle$ (i.e., after the colon). This information is then placed in the input stream in three parts: the $\langle name \rangle$, the $\langle signature \rangle$ and a logic token indicating if a colon was found (to differentiate variables from function names). The $\langle name \rangle$ does not include the escape character, and both the $\langle name \rangle$ and $\langle signature \rangle$ are made up of tokens with category code 12 (other).

The next three functions decompose T_EX macros into their constituent parts: if the $\langle token \rangle$ passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

`\cs_prefix_spec:N` ★ `\cs_prefix_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the applicable T_EX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves `\long` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

T_EXhackers note: The prefix can be empty, `\long`, `\protected` or `\protected\long` with backslash replaced by the current escape character.

`\cs_parameter_spec:N` ★ `\cs_parameter_spec:N` $\langle token \rangle$

New: 2022-06-24

If the $\langle token \rangle$ is a macro, this function leaves the primitive T_EX parameter specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_parameter_spec:N \next:nn
```

leaves `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

T_EXhackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

```
\cs_replacement_spec:N * \cs_replacement_spec:N <token>
```

```
\cs_replacement_spec:c *
```

If the $\langle token \rangle$ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves `x#1~y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

T_EXhackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

```
\use:n      * \use:n      {\group_1}
\use:nn     * \use:nn     {\group_1} {\group_2}
\use:nnn    * \use:nnn    {\group_1} {\group_2} {\group_3}
\use:nnnn   * \use:nnnn   {\group_1} {\group_2} {\group_3} {\group_4}
```

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

T_EXhackers note: The `\use:n` function is equivalent to L^AT_EX 2_ε's `\@firstofone`.

<code>\use_i:nn</code>	<code>* \use_i:nn {⟨arg₁⟩} {⟨arg₂⟩}</code>
<code>\use_ii:nn</code>	<code>* \use_i:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
<code>\use_i:nnn</code>	<code>* \use_i:nnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩}</code>
<code>\use_iii:nnn</code>	<code>* \use_i:nnnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩} {⟨arg₅⟩}</code>
<code>\use_iv:nnnn</code>	<code>* \use_i:nnnnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩} {⟨arg₅⟩} {⟨arg₆⟩}</code>
<code>\use_i:nnnn</code>	<code>* \use_i:nnnnnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩} {⟨arg₅⟩} {⟨arg₆⟩} {⟨arg₇⟩}</code>
<code>\use_ii:nnnn</code>	<code>* \use_i:nnnnnnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩} {⟨arg₅⟩} {⟨arg₆⟩} {⟨arg₇⟩}</code>
<code>\use_iii:nnnn</code>	<code>* {⟨arg₈⟩}</code>
<code>\use_iv:nnnn</code>	<code>* \use_i:nnnnnnnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩} {⟨arg₅⟩} {⟨arg₆⟩} {⟨arg₇⟩}</code>
<code>\use_i:nnnnn</code>	<code>* {⟨arg₈⟩} {⟨arg₉⟩}</code>
<code>\use_ii:nnnnn</code>	<code>*</code>
<code>\use_iii:nnnnn</code>	<code>*</code>
<code>\use_iv:nnnnn</code>	<code>*</code>
<code>\use_v:nnnnn</code>	<code>*</code>
<code>\use_i:nnnnnn</code>	<code>*</code>
<code>\use_ii:nnnnnn</code>	<code>*</code>
<code>\use_iii:nnnnnn</code>	<code>*</code>
<code>\use_iv:nnnnnn</code>	<code>*</code>
<code>\use_v:nnnnnn</code>	<code>*</code>
<code>\use_vi:nnnnnn</code>	<code>*</code>
<code>\use_i:nnnnnnn</code>	<code>*</code>
<code>\use_ii:nnnnnnn</code>	<code>*</code>
<code>\use_iii:nnnnnnn</code>	<code>*</code>
<code>\use_iv:nnnnnnn</code>	<code>*</code>
<code>\use_v:nnnnnnn</code>	<code>*</code>
<code>\use_vi:nnnnnnn</code>	<code>*</code>
<code>\use_vii:nnnnnnn</code>	<code>*</code>
<code>\use_i:nnnnnnnn</code>	<code>*</code>
<code>\use_ii:nnnnnnnn</code>	<code>*</code>
<code>\use_iii:nnnnnnnn</code>	<code>*</code>
<code>\use_iv:nnnnnnnn</code>	<code>*</code>
<code>\use_v:nnnnnnnn</code>	<code>*</code>
<code>\use_vi:nnnnnnnn</code>	<code>*</code>
<code>\use_vii:nnnnnnnn</code>	<code>*</code>
<code>\use_viii:nnnnnnnn</code>	<code>*</code>
<code>\use_i:nnnnnnnnn</code>	<code>*</code>
<code>\use_ii:nnnnnnnnn</code>	<code>*</code>
<code>\use_iii:nnnnnnnnn</code>	<code>*</code>
<code>\use_iv:nnnnnnnnn</code>	<code>*</code>
<code>\use_v:nnnnnnnnn</code>	<code>*</code>
<code>\use_vi:nnnnnnnnn</code>	<code>*</code>
<code>\use_vii:nnnnnnnnn</code>	<code>*</code>
<code>\use_viii:nnnnnnnnn</code>	<code>*</code>
<code>\use_ix:nnnnnnnnn</code>	<code>*</code>

<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

<code>\use_ii_i:nn</code>	★	<code>\use_ii_i:nn {⟨arg₁⟩} {⟨arg₂⟩}</code>

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>

<code>\use_none:nn</code>	★

<code>\use_none:nnn</code>	★

<code>\use_none:nnnn</code>	★

<code>\use_none:nnnnn</code>	★

<code>\use_none:nnnnnn</code>	★

<code>\use_none:nnnnnnn</code>	★

<code>\use_none:nnnnnnnn</code>	★

<code>\use_none:nnnnnnnnn</code>	★

TeXhackers note: These are equivalent to L^AT_EX 2_ε's `\@gobble`, `\@gobbletwo`, etc.

<code>\use:e</code>	★	<code>\use:e {⟨expandable tokens⟩}</code>

Updated: 2023-07-05 Fully expands the `⟨token list⟩` in an **e**-type manner, in which parameter character (usually `#`) need not be doubled, *and* the function remains fully expandable.

TeXhackers note: `\use:e` is a wrapper around the primitive `\expanded`. It requires two expansions to complete its action.

4.5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

<code>\use_none_delimit_by_q_nil:w</code>	★	<code>\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil</code>

<code>\use_none_delimit_by_q_stop:w</code>	★	<code>\use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop</code>

<code>\use_none_delimit_by_q_recursion_stop:w</code>	★	<code>\use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩ \q_recursion_stop</code>

Absorb the `⟨balanced text⟩` from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<code>\use_i_delimit_by_q_nil:nw</code>	<code>★ \use_i_delimit_by_q_nil:nw {\langle inserted tokens \rangle} \langle balanced text \rangle \q_nil</code>
<code>\use_i_delimit_by_q_stop:nw</code>	<code>★ \use_i_delimit_by_q_stop:nw {\langle inserted tokens \rangle} \langle balanced text \rangle</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>★ \q_stop</code>

	<code>\use_i_delimit_by_q_recursion_stop:nw {\langle inserted tokens \rangle} \langle balanced text \rangle \q_recursion_stop</code>

Absorb the $\langle balanced\ text \rangle$ from the input stream delimited by the marker given in the function name, leaving $\langle inserted\ tokens \rangle$ in the input stream for further processing.

4.6 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the $\langle true\ code \rangle$ or the $\langle false\ code \rangle$. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {\langle true code \rangle} {\langle false code \rangle}
```

a function that turns the first argument into a control sequence (since it’s marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with $\langle true\ code \rangle$ and/or $\langle false\ code \rangle$ are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF
{ \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl }
{\langle true code \rangle} {\langle false code \rangle}
```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain T_EX and L^AT_EX 2_ε. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

4.6.1 Tests on control sequences

<code>\cs_if_eq_p:NN</code>	★ <code>\cs_if_eq_p:NN</code> $\langle cs_1 \rangle$ $\langle cs_2 \rangle$
<code>\cs_if_eq_p:(Nc cN cc)</code>	★ <code>\cs_if_eq:NNTF</code> $\langle cs_1 \rangle$ $\langle cs_2 \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\cs_if_eq:NNTF</code>	★ Compares the definition of two $\langle control\ sequences \rangle$ and is logically <code>true</code> if they are
<code>\cs_if_eq:(Nc cN cc)TF</code>	★ the same, i.e., if they have exactly the same definition when examined with <code>\cs_show:N</code> .

<code>\cs_if_exist_p:N</code>	★ <code>\cs_if_exist_p:N</code> $\langle control\ sequence \rangle$
<code>\cs_if_exist_p:c</code>	★ <code>\cs_if_exist:NNTF</code> $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\cs_if_exist:NNTF</code>	★ Tests whether the $\langle control\ sequence \rangle$ is currently defined (whether as a function or
<code>\cs_if_exist:cTF</code>	★ another control sequence type), and its meaning is not the primitive <code>\relax</code> token. This

is different from `\if_cs_exist:N`, which evaluates to `true` if passed the token `\relax` as an argument.

<code>\cs_if_free_p:N</code>	★ <code>\cs_if_free_p:N</code> $\langle control\ sequence \rangle$
<code>\cs_if_free_p:c</code>	★ <code>\cs_if_free:NNTF</code> $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\cs_if_free:NNTF</code>	★ This test is the negation of the above <code>\cs_if_exist:NNTF</code> .
<code>\cs_if_free:cTF</code>	★

4.6.2 Primitive conditionals

The ε -T_EX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`, except for `\if:w`.

<code>\if_true:</code>	★ <code>\if_true:</code> $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_false:</code>	★ <code>\if_false:</code> $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\else:</code>	★ <code>\reverse_if:N</code> $\langle primitive\ conditional \rangle$
<code>\fi:</code>	★ <code>\if_true:</code> always executes $\langle true\ code \rangle$, while <code>\if_false:</code> always executes $\langle false\ code \rangle$.
<code>\reverse_if:N</code>	★ <code>\reverse_if:N</code> reverses any two-way primitive conditional. <code>\else:</code> and <code>\fi:</code> delimit the branches of the conditional. The function <code>\or:</code> is documented in l3int and used in case switches.

T_EXhackers note: `\if_true:` and `\if_false:` are equivalent to their corresponding T_EX primitive conditionals `\iftrue` and `\iffalse`; `\else:` and `\fi:` are the T_EX primitives `\else` and `\fi`; `\reverse_if:N` is the ε -T_EX primitive `\unless`.

`\if_meaning:w` ★ `\if_meaning:w <arg1> <arg2> <true code> \else: <false code> \fi:`

`\if_meaning:w` executes `<true code>` when `<arg1>` and `<arg2>` are the same, otherwise it executes `<false code>`. `<arg1>` and `<arg2>` could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

T_EXhackers note: This is the T_EX primitive `\ifx`.

`\if:w` ★ `\if:w <token(s)> <true code> \else: <false code> \fi:`

`\if_charcode:w` ★ `\if_catcode:w <token(s)> <true code> \else: <false code> \fi:`

`\if_catcode:w` ★ `\if_charcode:w` is an alternative name for `\if:w`. These conditionals expand `<token(s)>` until two unexpandable tokens `<token1>` and `<token2>` are found; any further tokens up to the next unbalanced `\else:` are the true branch, ending with `<true code>`. It is executed if the condition is fulfilled, otherwise `<false code>` is executed. You can omit `\else:` when just in front of `\fi:` and you can nest `\if... \else:... \fi:` constructs inside the true branch or the `<false code>`. With `\exp_not:N`, you can prevent the expansion of a token.

`\if_catcode:w` tests if `<token1>` and `<token2>` have the same category code whereas `\if:w` and `\if_charcode:w` test if they have the same character code.

T_EXhackers note: `\if:w` and `\if_charcode:w` are both the T_EX primitive `\if`. `\if_catcode:w` is the T_EX primitive `\ifcat`.

`\if_cs_exist:N` ★ `\if_cs_exist:N <cs> <true code> \else: <false code> \fi:`

`\if_cs_exist:w` ★ `\if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:`

Check if `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into the primitive `\relax` token. This can be useful when dealing with control sequences which cannot be entered as a single token.

T_EXhackers note: These are the T_EX primitives `\ifdefined` and `\ifcsname`.

`\if_mode_horizontal:` ★ `\if_mode_horizontal: <true code> \else: <false code> \fi:`

`\if_mode_vertical:` ★ Execute `<true code>` if currently in horizontal mode, otherwise execute `<false code>`.

`\if_mode_math:` ★ Similar for the other functions.

`\if_mode_inner:` ★

T_EXhackers note: These are the T_EX primitives `\ifhmode`, `\ifvmode`, `\ifmmode`, and `\ifinner`.

4.7 Starting a paragraph

`\mode_leave_vertical:` `\mode_leave_vertical:`

Ensures that `TEX` is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

T_EXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the L^AT_EX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

4.8 Debugging support

`\debug_on:n` `\debug_on:n {⟨comma-separated list⟩}`
`\debug_off:n` `\debug_off:n {⟨comma-separated list⟩}`

Updated: 2023-05-23 Turn on and off within a group various debugging code, some of which is also available as `expl3` load-time options. The items that can be used in the `⟨list⟩` are

- `check-assertions` that checks the invariants declared in `\debug_assert:nn` and friends;
- `check-declarations` that checks all `expl3` variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- `check-expressions` that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- `deprecation` that makes deprecated commands produce errors;
- `log-functions` that logs function definitions and variable declarations;
- `all` that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing.

`\debug_suspend:` `\debug_suspend: ... \debug_resume:`
`\debug_resume:`

Suppress (locally) errors and logging from `debug` commands, except for the `deprecation` errors. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance `l3cctab` and `l3coffins`.

<code>\debug_assert:nN</code> <code>\debug_assert:nn</code> <code>\debug_assert:nNn</code> <code>\debug_assert:nNe</code> <code>\debug_assert:nnn</code> <code>\debug_assert:nne</code>	<code>*</code> <code>*</code> <code>*</code> <code>*</code> <code>*</code> <code>*</code>	<code>\debug_assert:nN {<module>} <boolean></code> <code>\debug_assert:nn {<module>} {<boolean expression>}</code> <code>\debug_assert:nNn {<module>} <boolean> {<message text>}</code> <code>\debug_assert:nNe {<module>} {<boolean expression>} {<message text>}</code> Assert that the invariant specified by <code><boolean></code> or <code><boolean expression></code> is true at this point in the code.

New: 2026-03-06

When assertion checking has been enabled with `\debug_on:n {check-assertions}`, a violation of the invariant will produce an error with a message formed from `<module>` and the optional `<message text>`.

The invariant will *only* be evaluated when assertion checking has been enabled. Therefore, assertions can be used liberally throughout the code without significant performance impact outside debugging.

Chapter 5

The l3expan module

Argument expansion

This module provides generic methods for expanding $\TeX$ arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the $\LaTeX$ 3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

5.1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_....`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

5.2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

<code>\cs_generate_variant:Nn</code> <code>\cs_generate_variant:cn</code>	<code>\cs_generate_variant:Nn <parent control sequence> {<variant argument specifiers>}</code>

This function is used to define argument-specifier variants of the `<parent control sequence>` for L^AT_EX3 code-level macros. The `<parent control sequence>` is first separated into the `<base name>` and `<original argument specifier>`. The comma-separated list of `<variant argument specifiers>` is then used to define variants of the `<original argument specifier>` if these are not already defined; entries which correspond to existing functions are silently ignored. For each `<variant>` given, a function is created that expands its arguments as detailed and passes them to the `<parent control sequence>`. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function should only be applied if the `<parent control sequence>` is already defined. (This is only enforced if debugging support `check-declarations` is enabled.) If the `<parent control sequence>` is protected or if the `<variant>` involves any `x` argument, then the `<variant control sequence>` is also protected. The `<variant>` is created globally, as is any `\exp_args:N<variant>` function needed to carry out the expansion. There is no need to re-apply `\cs_generate_variant:Nn` after changing the definition of the parent function: the variant will always use the current definition of the parent. Providing variants repeatedly is safe as `\cs_generate_variant:Nn` will only create new definitions if there is not already one available.

Only `n` and `N` arguments can be changed to other types. The only allowed changes are

- `c` variant of an `N` parent;
- `o`, `V`, `v`, `f`, `e`, or `x` variant of an `n` parent;
- `N`, `n`, `T`, `F`, or `p` argument unchanged.

This means the `<parent>` of a `<variant>` form is always unambiguous, even in cases where both an `n`-type parent and an `N`-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

When creating variants for conditional functions, `\prg_generate_conditional_variant:Nnn` provides a convenient way of handling the related function set.

For backward compatibility it is currently possible to make `n`, `o`, `V`, `v`, `f`, `e`, or `x`-type variants of an `N`-type argument or `N` or `c`-type variants of an `n`-type argument. Both are deprecated. The first because passing more than one token to an `N`-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a `V`-type or `v`-type variant instead of `c`-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

`\exp_args_generate:n` `\exp_args_generate:n {⟨variant argument specifiers⟩}`

Defines `\exp_args:N⟨variant⟩` functions for each `⟨variant⟩` given in the comma list `{⟨variant argument specifiers⟩}`. Each `⟨variant⟩` should consist of the letters `N`, `c`, `n`, `V`, `v`, `o`, `f`, `e`, `x`, `p` and the resulting function is protected if the letter `x` appears in the `⟨variant⟩`. This is only useful for cases where `\cs_generate_variant:Nn` is not applicable.

5.3 Introducing the variants

The `V` type returns the value of a register, which can be one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by `(cs)name`, the `v` specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of `TEX`'s `\message` (in particular `#` needs not be doubled). It relies on the primitive `\expanded` hence is fast.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `e` or `x` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3 + 4` and pass the result `7` as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result `7`, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected for x-type. If you use **f** type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi`: itself!

It is important to note that both **f**- and **o**-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, **o**-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, **f**-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable.

It is usually best to keep the following in mind when using variant forms.

- Variants with **x**-type arguments (that are fully expanded before being passed to the **n**-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using **f** or **e** expansion.
- In contrast, **e** expansion (full expansion, almost like **x** except for the treatment of `#`) does not prevent variants from being expandable (if the base function is).
- Finally **f** expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because the speed of internal functions that expand the arguments of a base function depend on what needs doing with each argument and where this happens in the list of arguments:

- for fastest processing any **c**-type arguments should come first followed by all other modified arguments;
- unchanged **N**-type args that appear before modified ones have a small performance hit;
- unchanged **n**-type args that appear before modified ones have a relative larger performance hit.

5.4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

<code>\exp_args:Nc</code> *	<code>\exp_args:Nc <function> {<tokens>}</code>
<code>\exp_args:cc</code> *	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>). The <code><tokens></code> are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged. The <code>:cc</code> variant constructs the <code><function></code> name in the same manner as described for the <code><tokens></code>.
<code>\exp_args:No</code> *	<code>\exp_args:No <function> {<tokens>} ...</code>
	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>). The <code><tokens></code> are expanded once, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.
<code>\exp_args:Nv</code> *	<code>\exp_args:Nv <function> <variable></code>
	This function absorbs two arguments (the names of the <code><function></code> and the <code><variable></code>). The content of the <code><variable></code> are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.
<code>\exp_args:Nv</code> *	<code>\exp_args:Nv <function> {<tokens>}</code>
	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>). The <code><tokens></code> are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a <code><variable></code>. The content of the <code><variable></code> are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.
<code>\exp_args:Ne</code> *	<code>\exp_args:Ne <function> {<tokens>}</code>
	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>) and exhaustively expands the <code><tokens></code>. The result is inserted in braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.
<code>\exp_args:Nf</code> *	<code>\exp_args:Nf <function> {<tokens>}</code>
	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>). The <code><tokens></code> are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.
<code>\exp_args:Nx</code>	<code>\exp_args:Nx <function> {<tokens>}</code>
	This function absorbs two arguments (the <code><function></code> name and the <code><tokens></code>) and exhaustively expands the <code><tokens></code>. The result is inserted in braces into the input stream <i>after</i> reinsertion of the <code><function></code>. Thus the <code><function></code> may take more than one argument: all others are left unchanged.

5.5 Manipulating two arguments

<code>\exp_args:NNc</code>	*	<code>\exp_args:NNc</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
<code>\exp_args:NNo</code>	*	These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.			
<code>\exp_args:NNV</code>	*				
<code>\exp_args:NNv</code>	*				
<code>\exp_args:NNe</code>	*				
<code>\exp_args:NNf</code>	*				
<code>\exp_args:Ncc</code>	*				
<code>\exp_args:Nco</code>	*				
<code>\exp_args:NcV</code>	*				
<code>\exp_args:Ncv</code>	*				
<code>\exp_args:Ncf</code>	*				
<code>\exp_args:NVV</code>	*				

<code>\exp_args:Nnc</code>	*	<code>\exp_args:Nnc</code>	$\langle token \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
<code>\exp_args:Nno</code>	*	These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.			
<code>\exp_args:Nnf</code>	*				
<code>\exp_args:NnV</code>	*				
<code>\exp_args:Nnv</code>	*				
<code>\exp_args:Nne</code>	*				
<code>\exp_args:Nce</code>	*				
<code>\exp_args:Noc</code>	*				
<code>\exp_args:Noo</code>	*				
<code>\exp_args:Nof</code>	*				
<code>\exp_args:Nfo</code>	*				
<code>\exp_args:Nff</code>	*				
<code>\exp_args:NVo</code>	*				
<code>\exp_args:Nee</code>	*				

<code>\exp_args:NNx</code>	*	<code>\exp_args:NNx</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens \rangle\}$
<code>\exp_args:Ncx</code>	*	These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.			
<code>\exp_args:Nnx</code>	*				
<code>\exp_args:Nox</code>	*				
<code>\exp_args:Nxo</code>	*				
<code>\exp_args:Nxx</code>	*				

5.6 Manipulating three arguments

<code>\exp_args:NNNo</code>	*	<code>\exp_args:NNNo</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\langle token_3 \rangle$	$\{\langle tokens \rangle\}$
<code>\exp_args:NNNV</code>	*	These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i>				
<code>\exp_args:NNNv</code>	*					
<code>\exp_args:NNNe</code>	*					
<code>\exp_args:Nccc</code>	*					
<code>\exp_args:NcNc</code>	*					
<code>\exp_args:NcNo</code>	*					
<code>\exp_args:Ncco</code>	*					

<code>\exp_args:NNno</code>	<code>*</code>	<code>\exp_args:NNno</code>	<code><token₁> <token₂> {<token₃>} {<tokens>}</code>
<code>\exp_args:NNnV</code>	<code>*</code>		
<code>\exp_args:NNnv</code>	<code>*</code>		
<code>\exp_args:NNne</code>	<code>*</code>		
<code>\exp_args:NNcc</code>	<code>*</code>		
<code>\exp_args:NNcf</code>	<code>*</code>		
<code>\exp_args:NNoo</code>	<code>*</code>		
<code>\exp_args:NNVV</code>	<code>*</code>		
<code>\exp_args:NNVv</code>	<code>*</code>		
<code>\exp_args:NNVe</code>	<code>*</code>		
<code>\exp_args:NNvV</code>	<code>*</code>		
<code>\exp_args:NNvv</code>	<code>*</code>		
<code>\exp_args:NNve</code>	<code>*</code>		
<code>\exp_args:NNeV</code>	<code>*</code>		
<code>\exp_args:NNev</code>	<code>*</code>		
<code>\exp_args:NNee</code>	<code>*</code>		
<code>\exp_args:NnNV</code>	<code>*</code>		
<code>\exp_args:Nnnnc</code>	<code>*</code>		
<code>\exp_args:Nnnno</code>	<code>*</code>		
<code>\exp_args:Nnnnf</code>	<code>*</code>		
<code>\exp_args:NnnV</code>	<code>*</code>		
<code>\exp_args:Nnnv</code>	<code>*</code>		
<code>\exp_args:Nnne</code>	<code>*</code>		
<code>\exp_args:Nnff</code>	<code>*</code>		
<code>\exp_args:Nnee</code>	<code>*</code>		
<code>\exp_args:Ncnc</code>	<code>*</code>		
<code>\exp_args:Ncno</code>	<code>*</code>		
<code>\exp_args:NcnV</code>	<code>*</code>		
<code>\exp_args:Ncnv</code>	<code>*</code>		
<code>\exp_args:Ncne</code>	<code>*</code>		
<code>\exp_args:Ncoo</code>	<code>*</code>		
<code>\exp_args:NcVV</code>	<code>*</code>		
<code>\exp_args:NcVv</code>	<code>*</code>		
<code>\exp_args:NcVe</code>	<code>*</code>		
<code>\exp_args:NcvV</code>	<code>*</code>		
<code>\exp_args:Ncvv</code>	<code>*</code>		
<code>\exp_args:Ncve</code>	<code>*</code>		
<code>\exp_args:NceV</code>	<code>*</code>		
<code>\exp_args:Ncev</code>	<code>*</code>		
<code>\exp_args:Ncee</code>	<code>*</code>		
<code>\exp_args:Nooo</code>	<code>*</code>		
<code>\exp_args:Noof</code>	<code>*</code>		
<code>\exp_args:Nffo</code>	<code>*</code>		
<code>\exp_args:NVNV</code>	<code>*</code>		
<code>\exp_args:Neee</code>	<code>*</code>		

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

<code>\exp_args:NNNx</code>	<code>\exp_args:NNNx</code>	<code>\langle token_1 \rangle \langle token_2 \rangle \langle tokens_1 \rangle \{\langle tokens_2 \rangle\}</code>
<code>\exp_args:NNnx</code>		
<code>\exp_args:NNox</code>		
<code>\exp_args:Nccx</code>		
<code>\exp_args:Ncnx</code>		
<code>\exp_args:Nnnx</code>		
<code>\exp_args:Nnox</code>		
<code>\exp_args:Noox</code>		

5.7 Unbraced expansion

<code>\exp_last_unbraced:No</code>	*	<code>\exp_last_unbraced:No</code>	<code>\langle token \rangle \{\langle tokens_1 \rangle\}</code>
<code>\exp_last_unbraced:NV</code>	*		
<code>\exp_last_unbraced:Nv</code>	*		
<code>\exp_last_unbraced:Ne</code>	*		
<code>\exp_last_unbraced:Nf</code>	*		
<code>\exp_last_unbraced:NNo</code>	*		
<code>\exp_last_unbraced:NNV</code>	*		
<code>\exp_last_unbraced:NNf</code>	*		
<code>\exp_last_unbraced:Nco</code>	*		
<code>\exp_last_unbraced:NcV</code>	*		
<code>\exp_last_unbraced:Nno</code>	*		
<code>\exp_last_unbraced:Nnf</code>	*		
<code>\exp_last_unbraced:Noo</code>	*		
<code>\exp_last_unbraced:Nfo</code>	*		
<code>\exp_last_unbraced:NNNo</code>	*		
<code>\exp_last_unbraced:NNNV</code>	*		
<code>\exp_last_unbraced:NNNf</code>	*		
<code>\exp_last_unbraced:NnNo</code>	*		
<code>\exp_last_unbraced:NNNNo</code>	*		
<code>\exp_last_unbraced:NNNNf</code>	*		

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is `f-` expanded.

<code>\exp_last_unbraced:Nx</code>	<code>\exp_last_unbraced:Nx</code>	<code>\langle function \rangle \{\langle tokens \rangle\}</code>

This function fully expands the `\langle tokens \rangle` and leaves the result in the input stream after reinsertion of the `\langle function \rangle`. This function is not expandable.

<code>\exp_last_two_unbraced:Noo</code>	*	<code>\exp_last_two_unbraced:Noo</code>	<code>\langle token \rangle \{\langle tokens_1 \rangle\} \{\langle tokens_2 \rangle\}</code>

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

`\exp_after:wN` ★ `\exp_after:wN <token1> <token2>`

Carries out a single expansion of `<token2>` (which may consume arguments) prior to the expansion of `<token1>`. If `<token2>` has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that `<token1>` may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal $\text{\TeX}$ category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_args:N<variant>` function.

$\text{\TeX}$ hackers note: This is the $\text{\TeX}$ primitive `\expandafter`.

5.8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

`\exp_not:N` ★ `\exp_not:N <token>`

Prevents expansion of the `<token>` in a context where it would otherwise be expanded, for example an **e**-type or **x**-type argument or the first token in an **o**-type or **f**-type argument.

$\text{\TeX}$ hackers note: This is the $\text{\TeX}$ primitive `\noexpand`. It only prevents expansion. At the beginning of an **f**-type argument, a space `<token>` is removed even if it appears as `\exp_not:N \c_space_token`. In an **e**-expanding definition (`\cs_new:Npe`), a macro parameter introduces an argument even if it appears as `\exp_not:N # 1`. This differs from `\exp_not:n`.

`\exp_not:c` ★ `\exp_not:c <tokens>`

Expands the `<tokens>` until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

`\exp_not:n` ★ `\exp_not:n <tokens>`

Prevents expansion of the `<tokens>` in an **e**-type or **x**-type argument. In all other cases the `<tokens>` continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

$\text{\TeX}$ hackers note: This is the ε - $\text{\TeX}$ primitive `\unexpanded`. In an **e**-expanding definition (`\cs_new:Npe`), `\exp_not:n {#1}` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`, and `\exp_not:n {#}` is equivalent to `#`, namely it inserts the character `#`.

`\exp_not:o` ★ `\exp_not:o <tokens>`

Expands the `<tokens>` once, then prevents any further expansion in **e**-type or **x**-type arguments using `\exp_not:n`.

	<code>\exp_not:V</code> *	<code>\exp_not:V</code> $\langle$ <i>variable</i> $\rangle$
		Recovers the content of the $\langle$ <i>variable</i> $\rangle$, then prevents expansion of this material in e-type or x-type arguments using <code>\exp_not:n</code> .
	<code>\exp_not:v</code> *	<code>\exp_not:v</code> $\{ \langle$ <i>tokens</i> $\rangle \}$
		Expands the $\langle$ <i>tokens</i> $\rangle$ until only characters remains, and then converts this into a control sequence which should be a $\langle$ <i>variable</i> $\rangle$ name. The content of the $\langle$ <i>variable</i> $\rangle$ is recovered, and further expansion in e-type or x-type arguments is prevented using <code>\exp_not:n</code> .
	<code>\exp_not:e</code> *	<code>\exp_not:e</code> $\{ \langle$ <i>tokens</i> $\rangle \}$
		Expands $\langle$ <i>tokens</i> $\rangle$ exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in e-type or x-type arguments using <code>\exp_not:n</code> . This is very rarely useful but is provided for consistency.
	<code>\exp_not:f</code> *	<code>\exp_not:f</code> $\{ \langle$ <i>tokens</i> $\rangle \}$
		Expands $\langle$ <i>tokens</i> $\rangle$ fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in e-type or x-type arguments using <code>\exp_not:n</code> .
	<code>\exp_stop_f:</code> *	<code>\foo_bar:f</code> $\{ \langle$ <i>tokens</i> $\rangle$ <code>\exp_stop_f:</code> $\langle$ <i>more tokens</i> $\rangle$ $\}$
		This function terminates an f-type expansion. Thus if a function <code>\foo_bar:f</code> starts an f-type expansion and all of $\langle$ <i>tokens</i> $\rangle$ are expandable <code>\exp_stop_f:</code> terminates the expansion of tokens even if $\langle$ <i>more tokens</i> $\rangle$ are also expandable. The function itself is an implicit space token. Inside an e-type or x-type expansion, it retains its form, but when typeset it produces the underlying space ($\sqcup$).

5.9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of $\text{\TeX}$ expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down $\text{\TeX}$ is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of $\langle$ *expandable-tokens* $\rangle$ as that will break badly if unexpandable tokens are encountered in that place!

<code>\exp:w</code>	★ <code>\exp:w <expandable tokens> \exp_end:</code>
<code>\exp_end:</code>	★ Expands <code><expandable-tokens></code> until reaching <code>\exp_end:</code> at which point expansion stops. The full expansion of <code><expandable tokens></code> has to be empty. If any token in <code><expandable tokens></code> or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result <code>\exp_end:</code> will be misinterpreted later on. ³

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of `<expandable-tokens>` rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

T_EXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the `<expandable tokens>`, but this should not be relied upon.

<code>\exp:w</code>	★ <code>\exp:w <expandable-tokens> \exp_end_continue_f:w <further-tokens></code>
<code>\exp_end_continue_f:w</code>	★ Expands <code><expandable-tokens></code> until reaching <code>\exp_end_continue_f:w</code> at which point expansion continues as an f-type expansion expanding <code><further-tokens></code> until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by <code>\exp_stop_f:</code>). As with all f-type expansions a space ending the expansion gets removed.

The full expansion of `<expandable-tokens>` has to be empty. If any token in `<expandable-tokens>` or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.⁴

In typical use cases `<expandable-tokens>` contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

```
\exp:w <expandable-tokens> \exp_end:
```

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

```
\exp:w \exp_end_continue_f:w <expandable-tokens> \exp_stop_f:
```

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

³Due to the implementation you might get the character in position 0 in the current font (typically “”) in the output without any error message!

<code>\exp:w</code>	★ <code>\exp:w</code> <i><expandable-tokens></i> <code>\exp_end_continue_f:nw</code> <i><further-tokens></i>
<code>\exp_end_continue_f:nw</code>	★

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If *<further-tokens>* starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

5.10 Internal functions

<code>\::n</code>	<code>\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }</code>
<code>\::N</code>	Internal forms for the base expansion types. These names do <i>not</i> conform to the general
<code>\::P</code>	L ^A T _E X3 approach as this makes them more readily visible in the log and so forth. They
<code>\::c</code>	should not be used outside this module.
<code>\::o</code>	
<code>\::e</code>	
<code>\::f</code>	
<code>\::x</code>	
<code>\::v</code>	
<code>\::V</code>	
<code>\:::</code>	

<code>\::o_unbraced</code>	<code>\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }</code>
<code>\::e_unbraced</code>	Internal forms for the expansion types which leave the terminal argument unbraced.
<code>\::f_unbraced</code>	These names do <i>not</i> conform to the general L ^A T _E X3 approach as this makes them more
<code>\::x_unbraced</code>	readily visible in the log and so forth. They should not be used outside this module.
<code>\::v_unbraced</code>	
<code>\::V_unbraced</code>	

⁴In this particular case you may get a character into the output as well as an error message.

Chapter 6

The l3sort module

Sorting functions

6.1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
  \int_compare:nNnTF { #1 } > { #2 }
  { \sort_return_swapped: }
  { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *comparison code* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

T_EXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaT_EX), depending on what other packages are loaded.

Internally, the code from `l3sort` stores items in `\toks` registers allocated locally. Thus, the *comparison code* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

<code>\sort_return_same:</code>	<code>\seq_sort:Nn <seq var></code>
<code>\sort_return_swapped:</code>	<code>{ ... \sort_return_same: or \sort_return_swapped: ... }</code>

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items #1 and #2 to be compared.

Chapter 7

The l3tl-analysis module

Analyzing token lists

This module provides functions that are particularly useful in the `l3regex` module for mapping through a token list one $\langle token \rangle$ at a time (including begin-group/end-group tokens). For `\tl_analysis_map_inline:Nn` or `\tl_analysis_map_inline:nn`, the token list is given as an argument; the analogous function `\peek_analysis_map_inline:n` documented in `l3token` finds tokens in the input stream instead. In both cases the user provides $\langle inline\ code \rangle$ that receives three arguments for each $\langle token \rangle$:

- $\langle tokens \rangle$, which both `o`-expand and `e/x`-expand to the $\langle token \rangle$. The detailed form of $\langle tokens \rangle$ may change in later releases.
- $\langle char\ code \rangle$, a decimal representation of the character code of the $\langle token \rangle$, `-1` if it is a control sequence.
- $\langle catcode \rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token \rangle$ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing " $\langle catcode \rangle$ ".

In addition, there is a debugging function `\tl_analysis_show:n`, very similar to the `\ShowTokens` macro from the `ted` package.

<code>\tl_analysis_show:N</code>	<code>\tl_analysis_show:n {$\langle token\ list \rangle$}</code>
<code>\tl_analysis_show:n</code>	<code>\tl_analysis_log:n {$\langle token\ list \rangle$}</code>
<code>\tl_analysis_log:N</code>	Displays to the terminal (or log) the detailed decomposition of the $\langle token\ list \rangle$ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.
<code>\tl_analysis_log:n</code>	

New: 2021-05-11

<code>\tl_analysis_map_inline:nn</code>	<code>\tl_analysis_map_inline:nn {$\langle token\ list \rangle$} {$\langle inline\ function \rangle$}</code>
<code>\tl_analysis_map_inline:Nn</code>	Applies the $\langle inline\ function \rangle$ to each individual $\langle token \rangle$ in the $\langle token\ list \rangle$. The $\langle inline\ function \rangle$ receives three arguments as explained above. As all other mappings the mapping is done at the current group level, i.e., any local assignments made by the $\langle inline\ function \rangle$ remain in effect after the loop.

Updated: 2022-03-26

Chapter 8

The `l3regex` module

Regular expressions in `TEX`

The `l3regex` module provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

8.1 Syntax of regular expressions

8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `\_[^_]*\_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `\_.*?\_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+|-]?d+` matches an explicit integer with at most one sign.
- `[\+|-\_]*d+\_*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+|-\_]*(d+|d*\.\d+)\_*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+|-\_]*(d+|d*\.\d+)\_*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\_*` matches an explicit dimension with any unit that $\text{\TeX}$ knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+|-\_]*((?i)nan|inf|(d+|d*\.\d+)\_*(e[\+|-\_]*d+)?)\_*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+|-\_]*(d+|cC.)\_*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+|-\(\)*d+\.](\+|-*/[\+|-\(\)*d+\.)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

8.1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches a star character). Some escape sequences of the form backslash-letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (A–Z, a–z, 0–9) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into $\text{\TeX}$ under normal category codes. For instance, `\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{<regex>}` syntax (see below).

Any special character which appears at a place where its special behavior cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

8.1.3 Characters classes

Character properties.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^~I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^~I\^~J\^~L\^~M]`.

`\v` Any vertical space character, equivalent to `[\^J\^K\^L\^M]`. Note that `\^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, i.e., alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9\_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences. Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

`[^...]` Negative character class. Matches any token other than the specified characters.

`[x-y]` Within a character class, this denotes a range (can be used with escaped characters).

`[:<name>:]` Within a character class (one more set of brackets), this denotes the POSIX character class `<name>`, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

`[:^<name>:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, etc.) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[\^6-9]` are equivalent.

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

`?` 0 or 1, greedy.

`??` 0 or 1, lazy.

`*` 0 or more, greedy.

`*?` 0 or more, lazy.

`+` 1 or more, greedy.

`+?` 1 or more, lazy.

`{n}` Exactly n .

`{n,}` n or more, greedy.

`{n,}?` n or more, lazy.

`{n,m}` At least n , no more than m , greedy.

`{n,m}?` At least n , no more than m , lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

`A|B|C` Either one of A, B, or C, investigating A first.

`(...)` Capturing group.

`(?:...)` Non-capturing group.

`(?|...)` Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labeled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

8.1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;

- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{regex}` A control sequence whose csname matches the `<regex>`, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.⁵

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[L0][A-F]]` matches what $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO\*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{<var name>}` matches the exact contents (both character codes and category codes) of the variable `\<var name>`, which are obtained by applying `\exp_not:v{<var name>}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{1_tmpa_regex}D` matches the tokens A and

⁵This last example also captures “`abc`” as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression `\l_tmpa_regex`. This behaves as if a non-capturing group were surrounding `\l_tmpa_regex`, and any group contained in `\l_tmpa_regex` is converted to a non-capturing group. Quantifiers are supported.

For instance, if `\l_tmpa_regex` has value `B|C`, then `A\ur{\l_tmpa_regex}D` is equivalent to `A(?:B|C)D` (matching `ABD` or `ACD`) and not to `AB|CD` (matching `AB` or `CD`). To get the latter effect, it is simplest to use TeX's expansion machinery directly: if `\l_mymodule_BC_tl` contains `B|C` then the following two lines show the same result:

```
\regex_show:n { A \u{\l_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

8.1.6 Miscellaneous

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^` or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\l_tmpa_int` holding the value 1.

The option `(?i)` makes the match case insensitive (treating `A–Z` and `a–z` as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[\?-B]` is equivalent to `[\?@ABab]` (and differs from the much larger class `[\?-b]`), and `(?i)[^aeiou]` matches any character which is not a vowel. The `i` option has no effect on `\c{...}`, on `\u{...}`, on character properties, or on character classes, for instance it has no effect at all in `(?i)\u{\l_foo_tl}\d\d[:lower:]`.

8.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group (...); similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `\_` inserts a space (spaces are ignored when not escaped);

- `\a, \e, \f, \n, \r, \t, \xhh, \x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c{<category>}<character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for $\text{\TeX}$, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?1|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(e11--e1)(o,--o) w(or--o)(1d--1)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code régime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `\_` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code régime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<var name>}` allows to insert the contents of the variable with name `<var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{1_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in `\l_tmpa_tl` containing the percent character with category code 7 (superscript) and an active tilde character.

8.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

	<code>\regex_new:N</code>	<code>\regex_new:N <regex var></code>	Creates a new <code><regex var></code> or raises an error if the name is already taken. The declaration is global. The <code><regex var></code> is initially such that it never matches.
	<code>\regex_set:Nn</code> <code>\regex_gset:Nn</code>	<code>\regex_set:Nn <regex var> {<regex>}</code>	Stores a compiled version of the <code><regex></code> in the <code><regex var></code> . The assignment is local for <code>\regex_set:Nn</code> and global for <code>\regex_gset:Nn</code> . For instance, this function can be used as
		<pre>\regex_new:N \l_my_regex \regex_set:Nn \l_my_regex { my\ (simple\)? reg(ex ular\ expression) }</pre>	
	<code>\regex_const:Nn</code>	<code>\regex_const:Nn <regex var> {<regex>}</code>	Creates a new constant <code><regex var></code> or raises an error if the name is already taken. The value of the <code><regex var></code> is set globally to the compiled version of the <code><regex></code> .
	<code>\regex_show:N</code> <code>\regex_show:n</code> <code>\regex_log:N</code> <code>\regex_log:n</code>	<code>\regex_show:n {<regex>}</code> <code>\regex_log:n {<regex>}</code>	Displays in the terminal or writes in the log file (respectively) how <code>l3regex</code> interprets the <code><regex></code> . For instance, <code>\regex_show:n {\A X Y}</code> shows
	New: 2021-04-26 Updated: 2021-04-29	<pre>+--branch anchor at start (\A) char code 88 (X) +--branch char code 89 (Y)</pre>	

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

8.4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by `\regex_set:Nn`.

<code>\regex_if_match:nnTF</code>	<code>\regex_if_match:nnTF {<regex>} {<token list>} {<true code>} {<false code>}</code>
<code>\regex_if_match:nVTF</code>	
<code>\regex_if_match:NnTF</code>	Tests whether the <code><regex></code> matches any part of the <code><token list></code> . For instance,
<code>\regex_if_match:NVTF</code>	<code>\regex_if_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }</code>
	<code>\regex_if_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }</code>

New: 2025-05-14

leaves TRUE then FALSE in the input stream.

<code>\regex_count:nnN</code>	<code>\regex_count:nnN {<regex>} {<token list>} <integer></code>
<code>\regex_count:nVN</code>	
<code>\regex_count:NnN</code>	Sets <code><integer></code> within the current TeX group level equal to the number of times <code><regex></code>
<code>\regex_count:NVN</code>	appears in <code><token list></code> . The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbbb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

<code>\regex_match_case:nn</code>	<code>\regex_match_case:nnTF</code>
<code>\regex_match_case:nnTF</code>	{
	{<regex ₁ >} {<code case ₁ >}
	{<regex ₂ >} {<code case ₂ >}
	...
	{<regex _n >} {<code case _n >}
	} {<token list>}
	{<true code>} {<false code>}

New: 2022-01-10

Determines which of the `<regular expressions>` matches at the earliest point in the `<token list>`, and leaves the corresponding `<code>` followed by the `<true code>` in the input stream. If several `<regex>` match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the `<regex>` match, the `<false code>` is left in the input stream. Each `<regex>` can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the `<token list>`, each of the `<regex>` is searched in turn. If one of them matches then the corresponding `<code>` is used and everything else is discarded, while if none of the `<regex>` match at a given position then the next starting position is attempted. If none of the `<regex>` match anywhere in the `<token list>` then nothing is left in the input stream. Note that this differs from nested `\regex_if_match:nnTF` statements since all `<regex>` are attempted at each position rather than attempting to match `<regex1>` at every position before moving on to `<regex2>`.

8.5 Submatch extraction

<code>\regex_extract_once:nnN</code>	<code>\regex_extract_once:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_extract_once:nVN</code>	<code>\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
<code>\regex_extract_once:nnNTF</code>	
<code>\regex_extract_once:nVNTF</code>	Finds the first match of the <code><regex></code> in the <code><token list></code> . If it exists, the match is stored as the first item of the <code><seq var></code> , and further items are the contents of capturing groups, in the order of their opening parenthesis. The <code><seq var></code> is assigned locally. If there is no match, the <code><seq var></code> is cleared. The testing versions insert the <code><true code></code> into the input stream if a match was found, and the <code><false code></code> otherwise.
<code>\regex_extract_once:NnN</code>	
<code>\regex_extract_once:NVN</code>	
<code>\regex_extract_once:NnNTF</code>	
<code>\regex_extract_once:NVNTF</code>	

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n -th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n - 1)$ in functions such as `\regex_replace_once:nnN`.

<code>\regex_extract_all:nnN</code>	<code>\regex_extract_all:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_extract_all:nVN</code>	<code>\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
<code>\regex_extract_all:nnNTF</code>	
<code>\regex_extract_all:nVNTF</code>	Finds all matches of the <code><regex></code> in the <code><token list></code> , and stores all the submatch information in a single sequence (concatenating the results of multiple <code>\regex_extract_once:nnN</code> calls). The <code><seq var></code> is assigned locally. If there is no match, the <code><seq var></code> is cleared. The testing versions insert the <code><true code></code> into the input stream if a match was found, and the <code><false code></code> otherwise. For instance, assume that you type
<code>\regex_extract_all:NnN</code>	
<code>\regex_extract_all:NVN</code>	
<code>\regex_extract_all:NnNTF</code>	
<code>\regex_extract_all:NVNTF</code>	

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

<code>\regex_split:nnN</code>	<code>\regex_split:nnN {<regex>} {<token list>} <seq var></code>
<code>\regex_split:nVN</code>	<code>\regex_split:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}</code>
<code>\regex_split:nnNTF</code>	Splits the <code><token list></code> into a sequence of parts, delimited by matches of the <code><regex></code> .
<code>\regex_split:nVNTF</code>	If the <code><regex></code> has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to <code><seq var></code> is local. If no match is
<code>\regex_split:NnN</code>	found the resulting <code><seq var></code> has the <code><token list></code> as its sole item. If the <code><regex></code>
<code>\regex_split:NVN</code>	matches the empty token list, then the <code><token list></code> is split into single tokens. The
<code>\regex_split:NnNTF</code>	testing versions insert the <code><true code></code> into the input stream if a match was found, and
<code>\regex_split:NVNTF</code>	the <code><false code></code> otherwise. For example, after

```

\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }

```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

8.6 Replacement

<code>\regex_replace_once:nnN</code>	<code>\regex_replace_once:nnN {<regex>} {<replacement>} <t1 var></code>
<code>\regex_replace_once:nVN</code>	<code>\regex_replace_once:nnNTF {<regex>} {<replacement>} <t1 var> {<true code>} {<false code>}</code>
<code>\regex_replace_once:nnNTF</code>	
<code>\regex_replace_once:nVNTF</code>	Searches for the <code><regex></code> in the contents of the <code><t1 var></code> and replaces the first match with
<code>\regex_replace_once:NnN</code>	the <code><replacement></code> . In the <code><replacement></code> , <code>\0</code> represents the full match, <code>\1</code> represents
<code>\regex_replace_once:NVN</code>	the contents of the first capturing group, <code>\2</code> of the second, etc. The result is assigned
<code>\regex_replace_once:NnNTF</code>	locally to <code><t1 var></code> .
<code>\regex_replace_once:NVNTF</code>	

<code>\regex_replace_all:nnN</code>	<code>\regex_replace_all:nnN {<regex>} {<replacement>} <t1 var></code>
<code>\regex_replace_all:nVN</code>	<code>\regex_replace_all:nnNTF {<regex>} {<replacement>} <t1 var> {<true code>} {<false code>}</code>
<code>\regex_replace_all:nnNTF</code>	
<code>\regex_replace_all:nVNTF</code>	Replaces all occurrences of the <code><regex></code> in the contents of the <code><t1 var></code> by the
<code>\regex_replace_all:NnN</code>	<code><replacement></code> , where <code>\0</code> represents the full match, <code>\1</code> represents the contents of the
<code>\regex_replace_all:NVN</code>	first capturing group, <code>\2</code> of the second, etc. Every match is treated independently, and
<code>\regex_replace_all:NnNTF</code>	matches cannot overlap. The result is assigned locally to <code><t1 var></code> .
<code>\regex_replace_all:NVNTF</code>	

<code>\regex_replace_case_once:nN</code>	<code>\regex_replace_case_once:nNTF</code>
<code>\regex_replace_case_once:nNTF</code>	<pre> { {<regex₁>} {<replacement₁>} {<regex₂>} {<replacement₂>} ... {<regex_n>} {<replacement_n>} } <tl var> {<true code>} {<false code>} </pre>
New: 2022-01-10	

Replaces the earliest match of the regular expression $(?| \langle \text{regex}_1 \rangle | \dots | \langle \text{regex}_n \rangle)$ in the $\langle \text{tl var} \rangle$ by the $\langle \text{replacement} \rangle$ corresponding to which $\langle \text{regex}_i \rangle$ matched, then leaves the $\langle \text{true code} \rangle$ in the input stream. If none of the $\langle \text{regex} \rangle$ match, then the $\langle \text{tl var} \rangle$ is not modified, and the $\langle \text{false code} \rangle$ is left in the input stream. Each $\langle \text{regex} \rangle$ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the $\langle \text{token list} \rangle$, each of the $\langle \text{regex} \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle \text{replacement} \rangle$ as described for `\regex_replace_once:nnN`. This is equivalent to checking with `\regex_match_case:nn` which $\langle \text{regex} \rangle$ matches, then performing the replacement with `\regex_replace_once:nnN`.

<code>\regex_replace_case_all:nN</code>	<code>\regex_replace_case_all:nNTF</code>
<code>\regex_replace_case_all:nNTF</code>	<pre> { {<regex₁>} {<replacement₁>} {<regex₂>} {<replacement₂>} ... {<regex_n>} {<replacement_n>} } <tl var> {<true code>} {<false code>} </pre>
New: 2022-01-10	

Replaces all occurrences of all $\langle \text{regex} \rangle$ in the $\langle \text{token list} \rangle$ by the corresponding $\langle \text{replacement} \rangle$. Every match is treated independently, and matches cannot overlap. The result is assigned locally to $\langle \text{tl var} \rangle$, and the $\langle \text{true code} \rangle$ or $\langle \text{false code} \rangle$ is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the $\langle \text{token list} \rangle$, each of the $\langle \text{regex} \rangle$ is searched in turn. If one of them matches then it is replaced by the corresponding $\langle \text{replacement} \rangle$, and the search resumes at the position that follows this match (and replacement). For instance

```

\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
{
  { [A-Za-z]+ } { ‘\0’ }
  { \b } { --- }
  { . } { [\0] }
} \l_tmpa_tl

```

results in $\backslash\text{l\_tmpa\_tl}$ having the contents ‘Hello’---[,][$\backslash$][‘world’]---[!]. Note in particular that the word-boundary assertion $\backslash\text{b}$ did not match at the start of words because the case $[A-Za-z]^+$ matched at these positions. To change this, one could simply swap the order of the two cases in the argument of `\regex_replace_case_all:nN`.

8.7 Scratch regular expressions

<code>\l_tmpa_regex</code>	Scratch regex for local assignment. These are never used by the kernel code, and so are
<code>\l_tmpb_regex</code>	safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by
	other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_regex</code>	Scratch regex for global assignment. These are never used by the kernel code, and so
<code>\g_tmpb_regex</code>	are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by
	other non-kernel code and so should only be used for short-term storage.

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
Additional error-checking to come.
- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs.
Possibly update `\_regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `\_regex_action_free_group:n` only be used for greedy `{n,}` quantifier?
(I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than `\g__regex_balance_t1` to build the function `\__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `\__regex_action_free:n`.
- Optimize the use of `\__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better “show” for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: “if what follows is [...], then [...]”.
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break:` and then of playing well with `\tl_map_break:` called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?

- Named subpatterns: $\text{\TeX}$ programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- $\backslash\text{ddd}$, matching the character with octal code ddd : we already have $\backslash\text{x}\{\dots\}$ and the syntax is confusingly close to what we could have used for backreferences ($\backslash 1$, $\backslash 2$, ...), making it harder to produce useful error message.
- $\backslash\text{cx}$, similar to $\text{\TeX}$'s own $\backslash\text{^^}\text{x}$.
- Comments: $\text{\TeX}$ already has its own system for comments.
- $\backslash\text{Q}\dots\backslash\text{E}$ escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- $\backslash\text{C}$ single byte in UTF-8 mode: $\text{Xe}\text{\TeX}$ and $\text{Lua}\text{\TeX}$ serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Chapter 9

The l3prg module

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are *⟨true⟩* and *⟨false⟩*.

L^AT_EX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean *⟨true⟩* or *⟨false⟩*. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean *⟨true⟩* or *⟨false⟩* values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the N) and then executes either `true` or `false` depending on the result.

T_EXhackers note: The arguments are executed after exiting the underlying `\if... \fi:` structure.

9.1 Defining a set of conditional functions

<code>\prg_new_conditional:Npnn</code>	<code>\prg_new_conditional:Npnn \<name>:\<arg spec> \<parameters> {\<conditions>}</code>
<code>\prg_set_conditional:Npnn</code>	<code>{\<code>}</code>
<code>\prg_gset_conditional:Npnn</code>	<code>\prg_new_conditional:Nnn \<name>:\<arg spec> {\<conditions>} {\<code>}</code>
<code>\prg_new_protected_conditional:Npnn</code>	
<code>\prg_set_protected_conditional:Npnn</code>	
<code>\prg_gset_protected_conditional:Npnn</code>	
<code>\prg_new_conditional:Nnn</code>	
<code>\prg_set_conditional:Nnn</code>	
<code>\prg_gset_conditional:Nnn</code>	
<code>\prg_new_protected_conditional:Nnn</code>	
<code>\prg_set_protected_conditional:Nnn</code>	
<code>\prg_gset_protected_conditional:Nnn</code>	

Updated: 2022-11-01

These functions create a family of conditionals using the same `<code>` to perform the test created. Those non-protected conditionals are expandable if `<code>` is. The **new** versions check for existing definitions and perform assignments globally (cf. `\cs_new:Npn`) whereas the **set** versions do no check and perform assignments locally (cf. `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of `<conditions>`, which should be one or more of T, F and TF, and for non-protected conditionals p. For public conditionals, a full set of forms should be provided: this contrasts with strictly internal conditionals, where only the required subset need be defined.

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\<name>_p:\<arg spec>` — a predicate function which will supply either a logical **true** or logical **false**. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for **protected** conditionals.
- `\<name>:\<arg spec>T` — a function with one more argument than the original `<arg spec>` demands. The `<true branch>` code in this additional argument will be left on the input stream only if the test is **true**.
- `\<name>:\<arg spec>F` — a function with one more argument than the original `<arg spec>` demands. The `<false branch>` code in this additional argument will be left on the input stream only if the test is **false**.
- `\<name>:\<arg spec>TF` — a function with two more argument than the original `<arg spec>` demands. The `<true branch>` code in the first additional argument will be left on the input stream if the test is **true**, while the `<false branch>` code in the second argument will be left on the input stream if the test is **false**.

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `<argument specification>` but this is not enforced. The **Nnn** versions infer the number of arguments from the argument specification given (cf. `\cs_new:Nn`, etc.). Within the `<code>`, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```

\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
  \prg_return_true:
\else:
  \prg_return_false:
\fi:
\fi:
}

```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNTF` (because `F` is missing from the `\conditions` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

The special case where the code of a conditional ends with `\prg_return_true:` `\else:` `\prg_return_false:` `\fi:` is optimized.

```

\prg_new_eq_conditional:NNn \prg_new_eq_conditional:NNn \<name_1>:\<arg spec> \<name_2>:\<arg spec> {\<conditions>}
\prg_set_eq_conditional:NNn
\prg_gset_eq_conditional:NNn

```

Updated: 2023-05-26

These functions copy a family of conditionals. The **new** version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the **set** version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of `\conditions`, which should be one or more of `p`, `T`, `F` and `TF`.

```

\prg_return_true: * \prg_return_true:
\prg_return_false: * \prg_return_false:

```

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an **f**-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

```

\prg_generate_conditional_variant:Nnn \prg_generate_conditional_variant:Nnn \<name>:\<arg spec> {\<variant
argument specifiers>} {\<condition specifiers>}

```

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn \<conditional> {\<variant argument specifiers>}` on each `\<conditional>` described by the `\<condition specifiers>`. These base-form `\<conditionals>` are obtained from the `\<name>` and `\<arg spec>` as described for `\prg_new_conditional:Npnn`, and they should be defined.

9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, etc. which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, etc., in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<code>\bool_new:N</code>	<code>\bool_new:N</code> $\langle boolean \rangle$

<code>\bool_new:c</code>	Creates a new $\langle boolean \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle boolean \rangle$ is initially <code>false</code> .

<code>\bool_const:Nn</code>	<code>\bool_const:Nn</code> $\langle boolean \rangle$ $\{ \langle boolexpr \rangle \}$

<code>\bool_const:cn</code>	Creates a new constant $\langle boolean \rangle$ or raises an error if the name is already taken. The value of the $\langle boolean \rangle$ is set globally to the result of evaluating the $\langle boolexpr \rangle$.

<code>\bool_set_false:N</code>	<code>\bool_set_false:N</code> $\langle boolean \rangle$

<code>\bool_set_false:c</code>	Sets $\langle boolean \rangle$ logically <code>false</code> .

<code>\bool_gset_false:N</code>	

<code>\bool_gset_false:c</code>	

<code>\bool_set_true:N</code>	<code>\bool_set_true:N</code> $\langle boolean \rangle$

<code>\bool_set_true:c</code>	Sets $\langle boolean \rangle$ logically <code>true</code> .

<code>\bool_gset_true:N</code>	

<code>\bool_gset_true:c</code>	

<code>\bool_set_eq:NN</code>	<code>\bool_set_eq:NN</code> $\langle boolean_1 \rangle$ $\langle boolean_2 \rangle$

<code>\bool_set_eq:(cN Nc cc)</code>	Sets $\langle boolean_1 \rangle$ to the current value of $\langle boolean_2 \rangle$.

<code>\bool_gset_eq:NN</code>	

<code>\bool_gset_eq:(cN Nc cc)</code>	

<code>\bool_set:Nn</code>	<code>\bool_set:Nn</code> $\langle boolean \rangle$ $\{ \langle boolexpr \rangle \}$

<code>\bool_set:cn</code>	Evaluates the $\langle boolean \text{ expression} \rangle$ as described for <code>\bool_if:nTF</code> , and sets the $\langle boolean \rangle$ variable to the logical truth of this evaluation.

<code>\bool_gset:Nn</code>	

<code>\bool_gset:cn</code>	

<code>\bool_set_inverse:N</code>	<code>\bool_set_inverse:N <boolean></code>
<code>\bool_set_inverse:c</code>	
<code>\bool_gset_inverse:N</code>	Toggles the <code><boolean></code> from <code>true</code> to <code>false</code> and conversely: sets it to the inverse of its
<code>\bool_gset_inverse:c</code>	current value.
<code>\bool_if_p:N</code>	<code>\bool_if_p:N <boolean></code>
<code>\bool_if_p:c</code>	<code>\bool_if:NTF <boolean> {<true code>} {<false code>}</code>
<code>\bool_if:NTF</code>	
<code>\bool_if:cTF</code>	Tests the current truth of <code><boolean></code> , and continues expansion based on this result.
<code>\bool_to_str:N</code>	<code>\bool_to_str:N <boolean></code>
<code>\bool_to_str:c</code>	<code>\bool_to_str:n {<boolean expression>}</code>
<code>\bool_to_str:n</code>	Expands to the string <code>true</code> or <code>false</code> depending on the logical truth of the <code><boolean></code> or
	<code><boolean expression></code> .
	New: 2021-11-01
	Updated: 2023-11-14
<code>\bool_show:N</code>	<code>\bool_show:N <boolean></code>
<code>\bool_show:c</code>	
	Displays the logical truth of the <code><boolean></code> on the terminal.
	Updated: 2021-04-29
<code>\bool_show:n</code>	<code>\bool_show:n {<boolean expression>}</code>
	Displays the logical truth of the <code><boolean expression></code> on the terminal.
<code>\bool_log:N</code>	<code>\bool_log:N <boolean></code>
<code>\bool_log:c</code>	
	Writes the logical truth of the <code><boolean></code> in the log file.
	Updated: 2021-04-29
<code>\bool_log:n</code>	<code>\bool_log:n {<boolean expression>}</code>
	Writes the logical truth of the <code><boolean expression></code> in the log file.
<code>\bool_if_exist_p:N</code>	<code>\bool_if_exist_p:N <boolean></code>
<code>\bool_if_exist_p:c</code>	<code>\bool_if_exist:NTF <boolean> {<true code>} {<false code>}</code>
<code>\bool_if_exist:NTF</code>	
<code>\bool_if_exist:cTF</code>	Tests whether the <code><boolean></code> is currently defined. This does not check that the <code><boolean></code>
	really is a boolean variable.

9.2.1 Constant and scratch booleans

<code>\c_true_bool</code>	Constants that represent <code>true</code> and <code>false</code> , respectively. Used to implement predicates.
<code>\c_false_bool</code>	
<code>\l_tmpa_bool</code>	A scratch boolean for local assignment. It is never used by the kernel code, and so is
<code>\l_tmpb_bool</code>	
	safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other
	non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_bool</code>	A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_bool</code>	

9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean `<true>` or `<false>` values, it seems only fitting that we also provide a parser for `<boolean expressions>`.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean `<true>` or `<false>`. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

T_EXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in T_EX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

```
\bool_if_p:n * \bool_if_p:n {<boolean expression>}
\bool_if:nTF * \bool_if:nTF {<boolean expression>} {<true code>} {<false code>}
```

Tests the current truth of `<boolean expression>`, and continues expansion based on this result. The `<boolean expression>` should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

```
\bool_lazy_all_p:n * \bool_lazy_all_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_all:nTF * \bool_lazy_all:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

Implements the “And” operation on the `<boolean expressions>`, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two `<boolean expressions>`.

```
\bool_lazy_and_p:nn * \bool_lazy_and_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_and:nnTF * \bool_lazy_and:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the `<boolexpr2>` is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two `<boolean expressions>`.

```
\bool_lazy_any_p:n * \bool_lazy_any_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_any:nTF * \bool_lazy_any:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

Implements the “Or” operation on the `<boolean expressions>`, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two `<boolean expressions>`.

```
\bool_lazy_or_p:nn * \bool_lazy_or_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_or:nnTF * \bool_lazy_or:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

Implements the “Or” operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the `<boolexpr2>` is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two `<boolean expressions>`.

```
\bool_not_p:n * \bool_not_p:n {<boolean expression>}
```

Function version of `!(<boolean expression>)` within a boolean expression.

<code>\bool_xor_p:nn</code>	★	<code>\bool_xor_p:nn {<boolean>}{<boolean>}</code>
<code>\bool_xor:nnTF</code>	★	<code>\bool_xor:nnTF {<boolean>}{<boolean>}{<true code>}{<false code>}</code>

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn</code>	★	<code>\bool_do_until:Nn <boolean> {<code>}</code>
<code>\bool_do_until:cn</code>	★	

Places the `<code>` in the input stream for T_EX to process, and then checks the logical value of the `<boolean>`. If it is `false` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean>` is `true`.

<code>\bool_do_while:Nn</code>	★	<code>\bool_do_while:Nn <boolean> {<code>}</code>
<code>\bool_do_while:cn</code>	★	

Places the `<code>` in the input stream for T_EX to process, and then checks the logical value of the `<boolean>`. If it is `true` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean>` is `false`.

<code>\bool_until_do:Nn</code>	★	<code>\bool_until_do:Nn <boolean> {<code>}</code>
<code>\bool_until_do:cn</code>	★	

This function first checks the logical value of the `<boolean>`. If it is `false` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean>` is re-evaluated. The process then loops until the `<boolean>` is `true`.

<code>\bool_while_do:Nn</code>	★	<code>\bool_while_do:Nn <boolean> {<code>}</code>
<code>\bool_while_do:cn</code>	★	

This function first checks the logical value of the `<boolean>`. If it is `true` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean>` is re-evaluated. The process then loops until the `<boolean>` is `false`.

<code>\bool_do_until:nn</code>	★	<code>\bool_do_until:nn {<boolean expression>}{<code>}</code>

Places the `<code>` in the input stream for T_EX to process, and then checks the logical value of the `<boolean expression>` as described for `\bool_if:nTF`. If it is `false` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean expression>` evaluates to `true`.

<code>\bool_do_while:nn</code>	★	<code>\bool_do_while:nn {<boolean expression>}{<code>}</code>

Places the `<code>` in the input stream for T_EX to process, and then checks the logical value of the `<boolean expression>` as described for `\bool_if:nTF`. If it is `true` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean expression>` evaluates to `false`.

<code>\bool_until_do:nn</code>	★	<code>\bool_until_do:nn {<boolean expression>}{<code>}</code>

This function first checks the logical value of the `<boolean expression>` (as described for `\bool_if:nTF`). If it is `false` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean expression>` is re-evaluated. The process then loops until the `<boolean expression>` is `true`.

```
\bool_while_do:nn ☆ \bool_while_do:nn {<boolean expression>} {<code>}
```

This function first checks the logical value of the *<boolean expression>* (as described for *\bool_if:nTF*). If it is *true* the *<code>* is placed in the input stream and expanded. After the completion of the *<code>* the truth of the *<boolean expression>* is re-evaluated. The process then loops until the *<boolean expression>* is *false*.

```
\bool_case:n ☆ \bool_case:nTF
\bool_case:nTF ☆ {
  {<boolexpr case1>} {<code case1>}
  {<boolexpr case2>} {<code case2>}
  ...
  {<boolexpr casen>} {<code casen>}
}
{<true code>}
{<false code>}
```

New: 2023-05-03

Evaluates in turn each of the *<boolean expression case>*s until the first one that evaluates to *true*. The *<code>* associated to this first case is left in the input stream, followed by the *<true code>*, and other cases are discarded. If none of the cases match then only the *<false code>* is inserted. The function *\bool_case:n*, which does nothing if there is no match, is also available. For example

```
\bool_case:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
    { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
    { Many }
  { \l__mypkg_special_bool }
    { Special }
}
{ No idea! }
```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

9.5 Producing multiple copies

```
\prg_replicate:nn ☆ \prg_replicate:nn {<integer expression>} {<tokens>}
```

Evaluates the *<integer expression>* (which should be zero or positive) and creates the resulting number of copies of the *<tokens>*. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

9.6 Detecting T_EX’s mode

```
\mode_if_horizontal_p: ☆ \mode_if_horizontal_p:
\mode_if_horizontal:TF ☆ \mode_if_horizontal:TF {<true code>} {<false code>}
```

Detects if T_EX is currently in horizontal mode.

```
\mode_if_inner_p: * \mode_if_inner_p:
\mode_if_inner:TF * \mode_if_inner:TF {\true code} {\false code}
```

Detects if T_EX is currently in inner mode.

```
\mode_if_math_p: * \mode_if_math_p:
\mode_if_math:TF * \mode_if_math:TF {\true code} {\false code}
```

Detects if T_EX is currently in maths mode.

```
\mode_if_vertical_p: * \mode_if_vertical_p:
\mode_if_vertical:TF * \mode_if_vertical:TF {\true code} {\false code}
```

Detects if T_EX is currently in vertical mode.

9.7 Primitive conditionals

```
\if_predicate:w * \if_predicate:w <predicate> <true code> \else: <false code> \fi:
```

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the `<predicate>` but to make the coding clearer this should be done through `\if_bool:N`.)

```
\if_bool:N * \if_bool:N <boolean> <true code> \else: <false code> \fi:
```

This function takes a boolean variable and branches according to the result.

9.8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in expl3. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

```
\prg_break_point:Nn * \prg_break_point:Nn \<type>_map_break: {\code}
```

Used to mark the end of a recursion or mapping: the functions `\<type>_map_break:` and `\<type>_map_break:n` use this to break out of the loop (see `\prg_map_break:Nn` for how to set these up). After the loop ends, the `<code>` is inserted into the input stream. This occurs even if the break functions are *not* applied: `\prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

```
\prg_map_break:Nn * \prg_map_break:Nn \<type>_map_break: {\user code}
```

```
...
\prg_break_point:Nn \<type>_map_break: {\ending code}
```

Breaks a recursion in mapping contexts, inserting in the input stream the `<user code>` after the `<ending code>` for the loop. The function breaks loops, inserting their `<ending code>`, until reaching a loop with the same `<type>` as its first argument. This `\<type>_map_break:` argument must be defined; it is simply used as a recognizable marker for the `<type>`.

For types with mappings defined in the kernel, `\<type>_map_break:` and `\<type>_map_break:n` are defined as `\prg_map_break:Nn \<type>_map_break: {}` and the same with `{}` omitted.

9.8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

<code>\prg_break_point:</code>	<code>*</code>	This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursion: the function <code>\prg_break:n</code> uses this to break out of the loop.

<code>\prg_break:</code>	<code>*</code>	<code>\prg_break:n {<code>}</code> ... <code>\prg_break_point:</code>
<code>\prg_break:n</code>	<code>*</code>	Breaks a recursion which has no <code><ending code></code> and which is not a user-breakable mapping (see for instance implementation of <code>\int_step_function:nnnN</code>), and inserts the <code><code></code> in the input stream.

9.9 Internal programming functions

<code>\group_align_safe_begin:</code>	<code>*</code>	<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>	<code>*</code>	...

		<code>\group_align_safe_end:</code>

These functions are used to enclose material in a `TEX` alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as `TEX` uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` would result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

Chapter 10

The l3sys module

System/runtime functions

10.1 The name of the job

`\c_sys_jobname_str` Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This is the T_EX primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

10.2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

The date and time at which the current job was started: these are all reported as integers.

T_EXhackers note: Whilst the underlying T_EX primitives `\time`, `\day`, `\month`, and `\year` can be altered by the user, this interface to the time and date is intended to be the “real” values.

`\c_sys_timestamp_str` The timestamp for the current job: the format is as described for `\file_timestamp:n`.

New: 2023-08-27

10.3 Engine

```

\sys_if_engine luatex_p: * \sys_if_engine pdftex_p:
\sys_if_engine luatex:TF * \sys_if_engine pdftex:TF {\true code} {\false code}
\sys_if_engine pdftex_p: * Conditionals which allow engine-specific code to be used. The names follow naturally
\sys_if_engine pdftex:TF * from those of the engine binaries: note that the (u)ptex tests are for  $\varepsilon$ -pTeX and  $\varepsilon$ -upTeX
\sys_if_engine ptex_p: * as expl3 requires the  $\varepsilon$ -TeX extensions. Each conditional is true for exactly one supported
\sys_if_engine ptex:TF * engine. In particular, \sys_if_engine ptex_p: is true for  $\varepsilon$ -pTeX but false for  $\varepsilon$ -upTeX.
\sys_if_engine uptex_p: *
\sys_if_engine uptex:TF *
\sys_if_engine xetex_p: *
\sys_if_engine xetex:TF *

```

```

\sys_if_engine opentype_p: * \sys_if_engine opentype_p:
\sys_if_engine opentype:TF * \sys_if_engine opentype:TF {\true code} {\false code}

```

New: 2024-11-05

Conditional which allows functionality-specific code to be used. The test is true for engines which can use OpenType fonts and thus full Unicode typesetting. This tests for features not engine name, but currently is equivalent to requiring either XeTeX or LuaTeX.

TeXhackers note: The underlying test here checks for `\Umathcode`, which is used to implement OpenType math font typesetting. Any engine which should give a `true` result here needs to provide general Unicode support (accepting the full UTF-8 range for character codes), a mechanism to load system fonts and a suitable interface for math mode typesetting.

```

\c_sys_engine_str

```

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

```

\c_sys_engine_exec_str

```

The name of the standard executable for the current TeX engine given as a lower case string: one of `luatex`, `luahtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

New: 2020-08-20

```

\c_sys_engine_format_str

```

The name of the preloaded format for the current TeX run given as a lower case string: one of `lualatex` (or `dvilualatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for L^ATeX, similar names for plain TeX (except pdfTeX in DVI mode yields `etex`), and `cont-en` for ConTeXt (i.e., the `\fmtname`).

New: 2020-08-20

`\c_sys_engine_version_str` The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTeX and LuaTeX this is of the form

$\langle major \rangle . \langle minor \rangle . \langle revision \rangle$

For XeTeX, the form is

$\langle major \rangle . \langle minor \rangle$

For pTeX and upTeX, only releases since TeX Live 2018 make the data available, and the form is more complex, as it comprises the pTeX version, the upTeX version and the e-pTeX version.

$p \langle major \rangle . \langle minor \rangle . \langle revision \rangle - u \langle major \rangle . \langle minor \rangle - \langle epTeX \rangle$

where the *u* part is only present for upTeX.

`\sys_timer: *` `\sys_timer:`

New: 2021-05-12

Expands to the current value of the engine's timer clock, a non-negative integer. This function is only defined for engines with timer support. This command measures not just CPU time but real time (including time waiting for user input). The unit are scaled seconds (2^{-16} seconds).

10.4 Output format

In L^AT_EX, the output format may be set in the preamble: as such, expl3 delays setting the information here until either

- `\sys_ensure_backend:` or `\sys_load_backend:n` are used
- `\begin{document}` is reached

`\sys_if_output_dvi_p: *` `\sys_if_output_dvi_p:`

`\sys_if_output_dvi:TF *` `\sys_if_output_dvi:TF` $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$

`\sys_if_output_pdf_p: *` Conditionals which give the current output mode the TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasize the most appropriate case.

`\sys_if_output_pdf:TF *`

`\c_sys_output_str` The current output mode given as a lower case string: one of `dvi` or `pdf`.

10.5 Platform

<code>\sys_if_platform_unix_p:</code>	<code>*</code>	<code>\sys_if_platform_unix_p:</code>
<code>\sys_if_platform_unix:TF</code>	<code>*</code>	<code>\sys_if_platform_unix:TF</code> $\{\langle true\ code\rangle\}$ $\{\langle false\ code\rangle\}$
<code>\sys_if_platform_windows_p:</code>	<code>*</code>	
<code>\sys_if_platform_windows:TF</code>	<code>*</code>	

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, i.e., all Unix-like systems are `unix` (including Linux and MacOS).

<code>\c_sys_platform_str</code>	The current platform given as a lower case string: one of <code>unix</code> , <code>windows</code> or <code>unknown</code> .

10.6 Random numbers

<code>\sys_rand_seed:</code>	<code>*</code>	<code>\sys_rand_seed:</code>

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

<code>\sys_gset_rand_seed:n</code>	<code>\sys_gset_rand_seed:n</code> $\{\langle int\ expr\rangle\}$

Globally sets the seed for the engine's pseudo-random number generator to the $\langle integer\ expression\rangle$. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

T_EXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

10.7 Access to the shell

<code>\sys_get_shell:nnN</code>	<code>\sys_get_shell:nnN</code> $\{\langle shell\ command\rangle\}$ $\{\langle setup\rangle\}$ $\langle t1\ var\rangle$
<code>\sys_get_shell:nnNTF</code>	<code>\sys_get_shell:nnNTF</code> $\{\langle shell\ command\rangle\}$ $\{\langle setup\rangle\}$ $\langle t1\ var\rangle$ $\{\langle true\ code\rangle\}$ $\{\langle false\ code\rangle\}$

Defines $\langle t1\ var\rangle$ to the text returned by the $\langle shell\ command\rangle$. The $\langle shell\ command\rangle$ is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the $\langle setup\rangle$ argument, which is run just before running the $\langle shell\ command\rangle$ (in a group). If shell escape is disabled, the $\langle t1\ var\rangle$ will be set to `\q_no_value` in the non-branching version. Note that quote characters (") *cannot* be used inside the $\langle shell\ command\rangle$. The `\sys_get_shell:nnNTF` conditional inserts the $\langle true\ code\rangle$ if the shell is available and no quote is detected, and the $\langle false\ code\rangle$ otherwise.

Note: It is not possible to tell from T_EX if a command is allowed in restricted shell escape. If restricted escape is enabled, the `true` branch is taken: if the command is forbidden at this stage, a low-level T_EX error will arise.

`\c_sys_shell_escape_int` This variable exposes the internal triple of the shell escape status. The possible values are

- 0 Shell escape is disabled
- 1 Unrestricted shell escape is enabled
- 2 Restricted shell escape is enabled

`\sys_if_shell_p: *` `\sys_if_shell_p:`
`\sys_if_shell:TF *` `\sys_if_shell:TF` `{⟨true code⟩}` `{⟨false code⟩}`

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

`\sys_if_shell_unrestricted_p: *` `\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF *` `\sys_if_shell_unrestricted:TF` `{⟨true code⟩}` `{⟨false code⟩}`

Performs a check for whether *unrestricted* shell escape is enabled.

`\sys_if_shell_restricted_p: *` `\sys_if_shell_restricted_p:`
`\sys_if_shell_restricted:TF *` `\sys_if_shell_restricted:TF` `{⟨true code⟩}` `{⟨false code⟩}`

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:TF`.

`\sys_shell_now:n` `\sys_shell_now:n` `{⟨tokens⟩}`
`\sys_shell_now:e` Execute `⟨tokens⟩` through shell escape immediately.

`\sys_shell_shipout:n` `\sys_shell_shipout:n` `{⟨tokens⟩}`
`\sys_shell_shipout:e` Execute `⟨tokens⟩` through shell escape at shipout.

10.8 System queries

Some queries can be made about the file system, etc., without needing to use unrestricted shell escape. This is carried out using the script `l3sys-query`, which is documented separately. The wrappers here use this script, if available, to obtain system information that is not directly available within the T_EX run. Note that if restricted shell escape is disabled, no results can be obtained.

<code>\sys_get_query:nN</code>	<code>\sys_get_query:nN {<cmd>} <tl var></code>
<code>\sys_get_query:nnN</code>	<code>\sys_get_query:nnN {<cmd>} {<spec>} <tl var></code>
<code>\sys_get_query:nnnN</code>	<code>\sys_get_query:nnnN {<cmd>} {<options>} {<spec>} <tl var></code>
New: 2024-03-08	Sets the <code><tl var></code> to the information returned by the <code>l3sys-query <cmd></code> , potentially
Updated: 2024-04-08	supplying the <code><options></code> and <code><spec></code> to the query call. The valid <code><cmd></code> names are at
	present

- `pwd` Returns the present working directory
- `ls` Returns a directory listing, using the `<spec>` to select files and applying the `<options>` if given

The `<spec>` is likely to contain the wildcards `*` or `?`, and will automatically be passed to the script without shell expansion. In a glob is needed within the `<options>`, this will need to be protected from shell expansion using `'` tokens.

The `<spec>` and `<options>`, if given, are expanded fully before passing to the underlying script.

Spaces in the output are stored as active tokens, allowing them to be replaced by for example a visible space easily. Other non-letter characters in the ASCII range are set to category code 12. The category codes for characters out of the ASCII range are left unchanged: typically this will mean that with an 8-bit engine, accented values can be typeset directly whilst in Unicode engines, standard category code setup will apply.

If more than one line of text is returned by the `<cmd>`, these will be separated by character 13 (`^^M`) tokens of category code 12. In most cases, `\sys_split_query:nnnN` should be preferred when multi-line output is expected.

<code>\sys_split_query:nN</code>	<code>\sys_split_query:nN {<cmd>} <seq var></code>
<code>\sys_split_query:nnN</code>	<code>\sys_split_query:nnN {<cmd>} {<spec>} <seq var></code>
<code>\sys_split_query:nnnN</code>	<code>\sys_split_query:nnnN {<cmd>} {<options>} {<spec>} <seq var></code>
New: 2024-03-08	Works as described for <code>\sys_split_query:nnnN</code> , but sets the <code><seq var></code> to contain one
	entry for each line returned by <code>l3sys-query</code> . This function should therefore be preferred
	where multi-line return is expected, e.g. for the <code>ls</code> command.

10.9 Loading configuration data

<code>\sys_load_backend:n</code>	<code>\sys_load_backend:n {<backend>}</code>
	Loads the additional configuration file needed for backend support. If the <code><backend></code> is
	empty, the standard backend for the engine in use will be loaded. This command may
	only be used once.

<code>\sys_ensure_backend:</code>	<code>\sys_ensure_backend:</code>
New: 2022-07-29	Ensures that a backend has been loaded by calling <code>\sys_load_backend:n</code> if required.

<code>\c_sys_backend_str</code>	Set to the name of the backend in use by <code>\sys_load_backend:n</code> when issued. Possible values are

- `pdftex`
- `luatex`
- `xetex`
- `dvips`
- `dvipdfmx`
- `dvisvgm`

<code>\sys_load_debug:</code>	<code>\sys_load_debug:</code> Load the additional configuration file for debugging support.

10.9.1 Final settings

<code>\sys_finalize:</code>	<code>\sys_finalize:</code>
New: 2025-05-25	Finalizes all system-dependent functionality: required before loading a backend.

Chapter 11

The l3msg module

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behavior can be altered and messages can be entirely suppressed.

11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `\_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

Some authors may find the need to include spaces as `~` characters tedious. This can be avoided by locally resetting the category code of `_`.

```

\char_set_catcode_space:n { '\ }
\msg_new:nnn {foo} {bar}
  {Some message text using '#1' and usual message shorthands \{ \ \ \}.}
\char_set_catcode_ignore:n { '\ }

```

although in general this may be confusing; simply writing the messages using ~ characters is the method favored by the team.

<code>\msg_new:nnnn</code> <code>\msg_new:nnee</code> <code>\msg_new:nnn</code> <code>\msg_new:nne</code>	<code>\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}</code> Creates a <code><message></code> for a given <code><module></code> . The message is defined to first give <code><text></code> and then <code><more text></code> if the user requests it. If no <code><more text></code> is available then a standard text is given instead. Within <code><text></code> and <code><more text></code> four parameters (<code>#1</code> to <code>#4</code>) can be used: these will be supplied at the time the message is used. An error is raised if the <code><message></code> already exists.

<code>\msg_set:nnnn</code> <code>\msg_set:nnn</code>	<code>\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code> Sets up the text for a <code><message></code> for a given <code><module></code> . The message is defined to first give <code><text></code> and then <code><more text></code> if the user requests it. If no <code><more text></code> is available then a standard text is given instead. Within <code><text></code> and <code><more text></code> four parameters (<code>#1</code> to <code>#4</code>) can be used: these will be supplied at the time the message is used.

<code>\msg_if_exist_p:nn *</code> <code>\msg_if_exist:nnTF *</code>	<code>\msg_if_exist_p:nn {<module>} {<message>}</code> <code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code> Tests whether the <code><message></code> for the <code><module></code> is currently defined.

11.2 Customizable information for message modules

<code>\msg_module_name:n *</code>	<code>\msg_module_name:n {<module>}</code> Expands to the public name of the <code><module></code> as defined by <code>\g_msg_module_name_prop</code> (or otherwise leaves the <code><module></code> unchanged).

<code>\msg_module_type:n *</code>	<code>\msg_module_type:n {<module>}</code> Expands to the description which applies to the <code><module></code> , for example a <code>Package</code> or <code>Class</code> . The information here is defined in <code>\g_msg_module_type_prop</code> , and will default to <code>Package</code> if an entry is not present.

<code>\g_msg_module_name_prop</code>	Provides a mapping between the module name used for messages, and that for documentation.

<code>\g_msg_module_type_prop</code>	Provides a mapping between the module name used for messages, and that type of module. For example, for L ^A T _E X3 core messages, an empty entry is set here meaning that they are not described using the standard <code>Package</code> text.

11.3 Contextual information for messages

	<code>\msg_line_context: ☆ \msg_line_context:</code>
	Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text <code>on line</code> .
	<code>\msg_line_number: ★ \msg_line_number:</code>
	Prints the current line number when a message is given.
	<code>\msg_fatal_text:n ★ \msg_fatal_text:n {⟨module⟩}</code>
	Produces the standard text
	Fatal Package ⟨module⟩ Error
	This function can be redefined to alter the language in which the message is given, using <code>#1</code> as the name of the ⟨module⟩ to be included. Any redefinition <i>must</i> produce output containing the ⟨module⟩ name, and will affect all messages using the <code>expl3</code> mechanism.
	<code>\msg_critical_text:n ★ \msg_critical_text:n {⟨module⟩}</code>
	Produces the standard text
	Critical Package ⟨module⟩ Error
	This function can be redefined to alter the language in which the message is given, using <code>#1</code> as the name of the ⟨module⟩ to be included. Any redefinition <i>must</i> produce output containing the ⟨module⟩ name, and will affect all messages using the <code>expl3</code> mechanism.
	<code>\msg_error_text:n ★ \msg_error_text:n {⟨module⟩}</code>
	Produces the standard text
	Package ⟨module⟩ Error
	This function can be redefined to alter the language in which the message is given, using <code>#1</code> as the name of the ⟨module⟩ to be included. Any redefinition <i>must</i> produce output containing the ⟨module⟩ name, and will affect all messages using the <code>expl3</code> mechanism.
	<code>\msg_warning_text:n ★ \msg_warning_text:n {⟨module⟩}</code>
	Produces the standard text
	Package ⟨module⟩ Warning
	This function can be redefined to alter the language in which the message is given, using <code>#1</code> as the name of the ⟨module⟩ to be included. The ⟨type⟩ of ⟨module⟩ may be adjusted: <code>Package</code> is the standard outcome: see <code>\msg_module_type:n</code> . Any redefinition <i>must</i> produce output containing the ⟨module⟩ name, and will affect all messages using the <code>expl3</code> mechanism.

`\msg_info_text:n *` `\msg_info_text:n {⟨module⟩}`

Produces the standard text:

Package `⟨module⟩` Info

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the `⟨module⟩` to be included. The `⟨type⟩` of `⟨module⟩` may be adjusted: **Package** is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the `⟨module⟩` name, and will affect all messages using the `expl3` mechanism.

`\msg_see_documentation_text:n *` `\msg_see_documentation_text:n {⟨module⟩}`

Produces the standard text

See the `⟨module⟩` documentation for further information.

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the `⟨module⟩` to be included. The name of the `⟨module⟩` is produced using `\msg_module_name:n`.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the **e**-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as `\~` or `\\` are *not* available; instead one should use `\iow_char:N \~` and `\iow_newline:`, respectively. The following message classes exist:

- **fatal**, ending the `TEX` run;
- **critical**, ending the file being input;
- **error**, interrupting the `TEX` run without ending it;
- **warning**, written to terminal and log file, for important messages that may require corrections by the user;
- **note** (less common than **info**) for important information messages written to the terminal and log file;
- **info** for normal information messages written to the log file only;
- **term** and **log** for un-decorated messages written to the terminal and log file, or to the log file only;
- **none** for suppressed messages.

```

\msg_fatal:nnnnnn      \msg_fatal:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
\msg_fatal:nneeee      three} {\arg four}
\msg_fatal:nnnnn
\msg_fatal:(nneee|nnnee)
\msg_fatal:nnnn
\msg_fatal:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_fatal:nnn
\msg_fatal:(nnV|nne)
\msg_fatal:nn

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the $\text{\TeX}$ run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

```

\msg_critical:nnnnnn   \msg_critical:nnnnnn {\module} {\message} {\arg one} {\arg two}
\msg_critical:nneeee   {\arg three} {\arg four}
\msg_critical:nnnnn
\msg_critical:(nneee|nnnee)
\msg_critical:nnnn
\msg_critical:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_critical:nnn
\msg_critical:(nnV|nne)
\msg_critical:nn

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, $\text{\TeX}$ stops reading the current input file. This may halt the $\text{\TeX}$ run (if the current file is the main file) or may abort reading a sub-file.

$\text{\TeX}$ hackers note: The $\text{\TeX}$ `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```

\msg_error:nnnnnn     \msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg
\msg_error:nneeee     three} {\arg four}
\msg_error:nnnnn
\msg_error:(nneee|nnnee)
\msg_error:nnnn
\msg_error:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_error:nnn
\msg_error:(nnV|nne)
\msg_error:nn

```

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

<code>\msg_warning:nnnnnn</code>	<code>\msg_warning:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_warning:nneeee</code>	<code>three>} {<arg four>}</code>
<code>\msg_warning:nnnnn</code>	
<code>\msg_warning:(nneee nnnee)</code>	
<code>\msg_warning:nnnn</code>	
<code>\msg_warning:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_warning:nnn</code>	
<code>\msg_warning:(nnV nne)</code>	
<code>\msg_warning:nn</code>	

Issues `<module>` warning `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The warning text is added to the log file and the terminal, but the T_EX run is not interrupted.

<code>\msg_note:nnnnnn</code>	<code>\msg_note:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_note:nneeee</code>	<code>three>} {<arg four>}</code>
<code>\msg_note:nnnnn</code>	<code>\msg_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_note:(nneee nnnee)</code>	<code>three>} {<arg four>}</code>
<code>\msg_note:nnnn</code>	
<code>\msg_note:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_note:nnn</code>	
<code>\msg_note:(nnV nne)</code>	
<code>\msg_note:nn</code>	
<code>\msg_info:nnnnnn</code>	
<code>\msg_info:nneeee</code>	
<code>\msg_info:nnnnn</code>	
<code>\msg_info:(nneee nnnee)</code>	
<code>\msg_info:nnnn</code>	
<code>\msg_info:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_info:nnn</code>	
<code>\msg_info:(nnV nne)</code>	
<code>\msg_info:nn</code>	

New: 2021-05-18

Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. For the more common `\msg_info:nnnnnn`, the information text is added to the log file only, while `\msg_note:nnnnnn` adds the info text to both the log file and the terminal. The T_EX run is not interrupted.

<code>\msg_term:nnnnnn</code>	<code>\msg_term:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_term:nneeee</code>	<code>three>} {<arg four>}</code>
<code>\msg_term:nnnnn</code>	<code>\msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_term:(nneee nnnee)</code>	<code>three>} {<arg four>}</code>
<code>\msg_term:nnnn</code>	
<code>\msg_term:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_term:nnn</code>	
<code>\msg_term:(nnV nne)</code>	
<code>\msg_term:nn</code>	
<code>\msg_log:nnnnnn</code>	
<code>\msg_log:nneeee</code>	
<code>\msg_log:nnnnn</code>	
<code>\msg_log:(nneee nnnee)</code>	
<code>\msg_log:nnnn</code>	
<code>\msg_log:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_log:nnn</code>	
<code>\msg_log:(nnV nne)</code>	
<code>\msg_log:nn</code>	

Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The output is briefer than `\msg_info:nnnnnn`, omitting for instance the module name. It is added to the log file by `\msg_log:nnnnnn` while `\msg_term:nnnnnn` also prints it on the terminal.

<code>\msg_none:nnnnnn</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_none:nneeee</code>	<code>three>} {<arg four>}</code>
<code>\msg_none:nnnnn</code>	
<code>\msg_none:(nneee nnnee)</code>	
<code>\msg_none:nnnn</code>	
<code>\msg_none:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_none:nnn</code>	
<code>\msg_none:(nnV nne)</code>	
<code>\msg_none:nn</code>	

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

11.4.1 Messages for showing material

<code>\msg_show:nnnnnn</code>	<code>\msg_show:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>\msg_show:nneeee</code>	<code>three)} {<arg four>}</code>
<code>\msg_show:nnnnn</code>	
<code>\msg_show:(nneee nnnee)</code>	
<code>\msg_show:nnnn</code>	
<code>\msg_show:(nnVV nnVn nnnV nnee nnne)</code>	
<code>\msg_show:nnn</code>	
<code>\msg_show:(nnV nne)</code>	
<code>\msg_show:nn</code>	

Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The information text is shown on the terminal and the $\text{\TeX}$ run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

<code>\msg_show_item:n</code>	<code>* \seq_map_function:NN <seq var> \msg_show_item:n</code>
<code>\msg_show_item_unbraced:n</code>	<code>* \prop_map_function:NN <property list> \msg_show_item:nn</code>
<code>\msg_show_item:nn</code>	<code>*</code>
<code>\msg_show_item_unbraced:nn</code>	<code>*</code>

Used in the text of messages for `\msg_show:nnnnnn` to show or log a list of items or key-value pairs. The output of `\msg_show_item:n` produces a newline, the prefix `>`, two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using `uu=>uu`, and the `unbraced` versions don't print the surrounding braces.

These functions are suitable for usage with iterator functions like `\seq_map_function:NN`, `\prop_map_function:NN`, etc. For example, with a sequence `\l_tmpa_seq` containing `a`, `{b}` and `\c`,

```
\seq_map_function:NN \l_tmpa_seq \msg_show_item:n
```

would expand to three lines:

```
>uu{a}
>uu{{b}}
>uu{\c}
```

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message

is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

```

\msg_expandable_error:nnnnnn      * \msg_expandable_error:nnnnnn {\module} {\message} {\arg one} {\arg
\msg_expandable_error:(nneeee|nnffff) * two} {\arg three} {\arg four}
\msg_expandable_error:nnnnnn      *
\msg_expandable_error:(nneeee|nnffff) *
\msg_expandable_error:nnnnnn      *
\msg_expandable_error:(nnee|nnff)  *
\msg_expandable_error:nnnnnn      *
\msg_expandable_error:(nne|nnf)    *
\msg_expandable_error:nnnnnn      *
\msg_expandable_error:nnnnnn      *

```

Issues an “Undefined error” message from T_EX itself using the undefined control sequence `\??? then prints “! <module>: <error message>”, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.`

11.5 Redirecting messages

Each message has a “name”, which can be used to alter the behavior of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some-more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behavior with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

```
\msg_redirect_class:nn \msg_redirect_class:nn {<class one>} {<class two>}
```

Changes the behavior of messages of *<class one>* so that they are processed using the code for those of *<class two>*. Each *<class>* can be one of **fatal**, **critical**, **error**, **warning**, **note**, **info**, **term**, **log**, **none**.

```
\msg_redirect_module:nnn \msg_redirect_module:nnn {<module>} {<class one>} {<class two>}
```

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the **warning** messages of *<module>* could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

```
\msg_redirect_name:nnn \msg_redirect_name:nnn {<module>} {<message>} {<class>}
```

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Chapter 12

The l3file module

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files $\text{\TeX}$ attempts to locate them using both the operating system path and entries in the $\text{\TeX}$ file database (most $\text{\TeX}$ systems use such a database). Thus the “current path” for $\text{\TeX}$ is somewhat broader than that for other programs.

For functions which expect a $\langle file\ name \rangle$ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (") are not permitted in file names as they are reserved for internal use by some $\text{\TeX}$ primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As $\text{\TeX}$ engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in $\text{\LaTeX}3$. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

<code>\ior_new:N</code>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\ior_new:N <stream></code>
<code>\iow_new:N</code>	Globally reserves the name of the <code><stream></code> , either for reading or for writing as appropriate. The <code><stream></code> is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a <code><stream></code> which has not been opened is an error, and the <code><stream></code> will behave as the corresponding <code>\c_term_....</code>
<code>\iow_new:c</code>	
<code>\ior_open:Nn</code>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until an <code>\ior_close:N</code> instruction is given or the TeX run ends. If the file is not found, an error is raised.
<code>\ior_open:NnTF</code>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code>	Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until a <code>\ior_close:N</code> instruction is given or the TeX run ends. The <code><true code></code> is then inserted into the input stream. If the file is not found, no error is raised and the <code><false code></code> is inserted into the input stream.
<code>\iow_open:Nn</code>	<code>\iow_open:Nn <stream> {<file name>}</code>
<code>\iow_open:(NV cn cV)</code>	Opens <code><file name></code> for writing using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until a <code>\iow_close:N</code> instruction is given or the TeX run ends. Opening a file for writing clears any existing content in the file (i.e., writing is <i>not</i> additive).
<code>\ior_shell_open:Nn</code>	<code>\ior_shell_open:Nn <stream> {<shell command>}</code>
	Opens the <i>pseudo</i> -file created by the output of the <code><shell command></code> for reading using <code><stream></code> as the control sequence for access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><shell command></code> until a <code>\ior_close:N</code> instruction is given or the TeX run ends. If piped system calls are disabled an error is raised. For details of handling of the <code><shell command></code> , see <code>\sys_get_shell:nnNTF</code> .
<code>\iow_shell_open:Nn</code>	<code>\iow_shell_open:Nn <stream> {<shell command>}</code>
New: 2023-05-25	Opens the <i>pseudo</i> -file created by the output of the <code><shell command></code> for writing using <code><stream></code> as the control sequence for access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><shell command></code> until an <code>\iow_close:N</code> instruction is given or the TeX run ends. If piped system calls are disabled an error is raised. For details of handling of the <code><shell command></code> , see <code>\sys_get_shell:nnNTF</code> .

```
\ior_close:N \ior_close:N <stream>
\ior_close:c \iow_close:N <stream>
\iow_close:N Closes the <stream>. Streams should always be closed when they are finished with as
\iow_close:c this ensures that they remain available to other programmers.
```

```
\ior_show:N \ior_show:N <stream>
\ior_show:c \ior_log:N <stream>
\ior_log:N \iow_show:N <stream>
\ior_log:c \iow_log:N <stream>
\iow_show:N Display (to the terminal or log file) the file name associated to the (read or write)
\iow_show:c <stream>.
\iow_log:N
\iow_log:c
```

New: 2021-05-11

```
\ior_show_list: \ior_show_list:
\ior_log_list: \ior_log_list:
\iow_show_list: \iow_show_list:
\iow_log_list: \iow_log_list:
```

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get:NN` and `\ior_str_get:NN`, and their branching equivalents, are designed to work with *files*.

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> <tl var></code>
<code>\ior_get:NNTF</code>	<code>\ior_get:NNTF <stream> <tl var> {<true code>} {<false code>}</code>

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material read from the `<stream>` is tokenized by $\text{\TeX}$ according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character `%` have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a b c`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \g_my_ior \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NMF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

$\text{\TeX}$ hackers note: This protected macro is a wrapper around the $\text{\TeX}$ primitive `\read`. Regardless of settings, $\text{\TeX}$ replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

<code>\ior_str_get:NN</code>	<code>\ior_str_get:NN <stream> <tl var></code>
<code>\ior_str_get:NNTF</code>	<code>\ior_str_get:NNTF <stream> <tl var> {<true code>} {<false code>}</code>

Function that reads one line from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material is read from the `<stream>` as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the `<tl var>` being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b c
```

results in a token list `a b c` with the letters `a`, `b`, and `c` having category code 12. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

$\text{\TeX}$ hackers note: This protected macro is a wrapper around the ε - $\text{\TeX}$ primitive `\readline`. Regardless of settings, $\text{\TeX}$ removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, i.e., any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

`\ior_map_inline:Nn` `\ior_map_inline:Nn <stream> {<inline function>}`

Applies the `<inline function>` to each set of `<lines>` obtained by calling `\ior_get:NN` until reaching the end of the file. T_EX ignores any trailing new-line marker from the file it reads. The `<inline function>` should consist of code which receives the `<line>` as #1.

`\ior_str_map_inline:Nn` `\ior_str_map_inline:Nn <stream> {<inline function>}`

Applies the `<inline function>` to every `<line>` in the `<stream>`. The material is read from the `<stream>` as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The `<inline function>` should consist of code which receives the `<line>` as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

`\ior_map_variable:NNn` `\ior_map_variable:NNn <stream> <tl var> {<code>}`

For each set of `<lines>` obtained by calling `\ior_get:NN` until reaching the end of the file, stores the `<lines>` in the `<tl var>` then applies the `<code>`. The `<code>` will usually make use of the `<variable>`, but this is not enforced. The assignments to the `<variable>` are local. Its value after the loop is the last set of `<lines>`, or its original value if the `<stream>` is empty. T_EX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

`\ior_str_map_variable:NNn` `\ior_str_map_variable:NNn <stream> <variable> {<code>}`

For each `<line>` in the `<stream>`, stores the `<line>` in the `<variable>` then applies the `<code>`. The material is read from the `<stream>` as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The `<code>` will usually make use of the `<variable>`, but this is not enforced. The assignments to the `<variable>` are local. Its value after the loop is the last `<line>`, or its original value if the `<stream>` is empty. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_str_map_inline:Nn`.

`\ior_map_break:` `\ior_map_break:`

Used to terminate a `\ior_map...` function before all lines from the $\langle stream \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \g_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario leads to low level $\text{\TeX}$ errors.

$\text{\TeX}$ hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n` `\ior_map_break:n { $\langle code \rangle$ }`

Used to terminate a `\ior_map...` function before all lines in the $\langle stream \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \g_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario leads to low level $\text{\TeX}$ errors.

$\text{\TeX}$ hackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N` $\star$ `\ior_if_eof_p:N` $\langle stream \rangle$
`\ior_if_eof:NTF` $\star$ `\ior_if_eof:NTF` $\langle stream \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests if the end of a file $\langle stream \rangle$ has been reached during a reading operation. The test also returns a `true` value if the $\langle stream \rangle$ is not open.

12.1.2 Reading from the terminal

<code>\ior_get_term:nN</code>	<code>\ior_get_term:nN {⟨prompt⟩} ⟨tl var⟩</code>
<code>\ior_str_get_term:nN</code>	Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the <code>⟨token list⟩</code> variable. Tokenization occurs as described for <code>\ior_get:NN</code> or <code>\ior_str_get:NN</code> , respectively. When the <code>⟨prompt⟩</code> is empty, <code>TeX</code> will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. <code>\iow_term:n</code> . Where the <code>⟨prompt⟩</code> is given, it will appear in the terminal followed by an <code>=</code> , e.g.
	<code>prompt=</code>

12.1.3 Writing to files

<code>\iow_now:Nn</code>	<code>\iow_now:Nn ⟨stream⟩ {⟨tokens⟩}</code>
<code>\iow_now:(NV Ne cn cV ce)</code>	This function writes <code>⟨tokens⟩</code> to the specified <code>⟨stream⟩</code> immediately (i.e., the write operation is called on expansion of <code>\iow_now:Nn</code>).
<code>\iow_log:n</code>	<code>\iow_log:n {⟨tokens⟩}</code>
<code>\iow_log:e</code>	This function writes the given <code>⟨tokens⟩</code> to the log (transcript) file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .
<code>\iow_show:n</code>	<code>\iow_show:n {⟨tokens⟩}</code>
<code>\iow_show:e</code>	This function writes the given <code>⟨tokens⟩</code> immediately to the same output as used by <code>\show</code> and <code>\showtokens</code> . At the start of a <code>TeX</code> run this will be the terminal, but may be redirected to a file if the primitive <code>\showstream</code> has been set.
<code>\iow_term:n</code>	<code>\iow_term:n {⟨tokens⟩}</code>
<code>\iow_term:e</code>	This function writes the given <code>⟨tokens⟩</code> to the terminal file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .
<code>\iow_shipout:Nn</code>	<code>\iow_shipout:Nn ⟨stream⟩ {⟨tokens⟩}</code>
<code>\iow_shipout:(Ne cn ce)</code>	This function writes <code>⟨tokens⟩</code> to the specified <code>⟨stream⟩</code> when the current page is finalized (i.e., at shipout). The <code>e</code> -type variants expand the <code>⟨tokens⟩</code> at the point where the function is used but <i>not</i> when the resulting tokens are written to the <code>⟨stream⟩</code> (cf. <code>\iow_shipout_e:Nn</code>).

TeXhackers note: At present, there is no `expl3` interface to set `\showstream`, but use of the `\iow_show:n` function is encouraged in places where direct writing to an I/O stream is intermixed with `show` functions.

TeXhackers note: When using `expl3` with a format other than `LATeX`, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<code>\iow_shipout_e:Nn</code>	<code>\iow_shipout_e:Nn <stream> {<tokens>}</code>
<code>\iow_shipout_e:(Ne cn ce)</code>	This function writes <code><tokens></code> to the specified <code><stream></code> when the current page is finalized (i.e., at shipout). The <code><tokens></code> are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalized during the page building process (such as the page number integer).
Updated: 2023-09-17	

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`. When using `expl3` with a format other than L^AT_EX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

<code>\iow_char:N</code> ★	<code>\iow_char:N \<char></code>
	Inserts <code><char></code> into the output stream. Useful when trying to write difficult characters such as %, {, }, etc. in messages, for example:

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

<code>\iow_newline:</code> ★	<code>\iow_newline:</code>
	Function to add a new line within the <code><tokens></code> written to a file. The function has no effect if writing is taking place without expansion (<i>e.g.</i> in the second argument of <code>\iow_now:Nn</code>).

T_EXhackers note: When using `expl3` with a format other than L^AT_EX, the character inserted by `\iow_newline:` is not recognized by T_EX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_e:Nn` and direct uses of primitive operations.

12.1.4 Wrapping lines in output

<code>\iow_wrap:nnnN</code>	<code>\iow_wrap:nnnN {<text>} {<run-on text>} {<set up>} <function></code>
<code>\iow_wrap:nenN</code>	

This function wraps the `<text>` to a fixed number of characters per line. At the start of each line which is wrapped, the `<run-on text>` is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the `<run-on text>` for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The `<text>` and `<run-on text>` are exhaustively expanded by the function, with the following substitutions:

- `\\` or `\iow_newline`: may be used to force a new line,
- `\_` may be used to represent a forced space (for example after a control sequence),
- `\#, \%, \{, \}, \~` may be used to represent the corresponding character,
- `\iow_wrap_allow_break`: may be used to allow a line-break without inserting a space,
- `\iow_indent:n` may be used to indent a part of the `<text>` (not the `<run-on text>`).

Additional functions may be added to the wrapping by using the `<set up>`, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the `<text>` which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, etc.

The result of the wrapping operation is passed as a braced argument to the `<function>`, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (i.e., the argument passed to the `<function>`) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an e-type expansion on the `<text>` to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` could be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the `<text>`.

<code>\iow_wrap_allow_break:</code>	<code>\iow_wrap_allow_break:</code>

New: 2023-04-25

In the first argument of `\iow_wrap:nnnN` (for instance in messages), inserts a break-point that allows a line break. If no break occurs, this function adds nothing to the output.

<code>\iow_indent:n</code>	<code>\iow_indent:n {<text>}</code>

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents `<text>` by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the `<text>`. In case the indented `<text>` should appear on separate lines from the surrounding text, use `\\` to force line breaks.

<code>\l_iow_line_count_int</code>	The maximum number of characters in a line to be written by the <code>\iow_wrap:nnn</code> function. This value depends on the T _E X system in use: the standard value is 78, which is typically correct for unmodified T _E X Live and MiK _T E _X systems.

12.1.5 Constant input–output streams, and variables

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\c_log_iow</code> <code>\c_term_iow</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.

<code>\g_tmpa_iow</code> <code>\g_tmpb_iow</code>	Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12.1.6 Primitive conditionals

<code>\if_eof:w</code> ★	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifeof`.

12.2 File operations

12.2.1 Basic file operations

<code>\g_file_curr_dir_str</code> <code>\g_file_curr_name_str</code> <code>\g_file_curr_ext_str</code>	Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (i.e., if it is in the T _E X search path), and does <i>not</i> end in / other than the case that it is exactly equal to the root directory. The <code><name></code> and <code><ext></code> parts together make up the file name, thus the <code><name></code> part may be thought of as the “job name” for the current file.

Note that T_EX does not provide information on the `<dir>` and `<ext>` part for the main (top level) file and that this file always has empty `<dir>` and `<ext>` components. Also, the `<name>` here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

<code>\l_file_search_path_seq</code>	Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and need not include the trailing slash. Spaces need not be quoted.
Updated: 2023-06-15	

TeXhackers note: When working as a package in L^AT_EX 2_ε, `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

<code>\file_if_exist_p:n</code> ☆	<code>\file_if_exist_p:n {<file name>}</code>
<code>\file_if_exist_p:V</code> ☆	<code>\file_if_exist:nTF {<file name>} {<true code>} {<false code>}</code>
<code>\file_if_exist:nTF</code> ☆	Expands the argument of the <code><file name></code> to give a string, then searches for this string using the current TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code> .
<code>\file_if_exist:VTF</code> ☆	
Updated: 2023-09-18	

Since TeX cannot remove files, only write to them, once a file has been found during a TeX run, it will exist until the end of the run unless a non-TeX process intervenes. Since file operations are relatively slow, `expl3` therefore internally tracks when a file is seen, and uses this information to avoid multiple filesystem checks. See `\file_forget:n` for how to indicate to `expl3` that a file may have been deleted *during* a TeX run, so that its presence in the filesystem can be reasserted with `\file_if_exist:nTF` and similar commands.

<code>\file_forget:n</code>	<code>\file_forget:n {<file name>}</code>
New: 2024-12-09	Resets the internal tracker for files such that a subsequent use of <code>\file_if_exist:nTF</code> , <code>\file_size:n</code> , etc., for the <code><file name></code> will re-query the filesystem rather than use any cached information. This can be used whether or not the file has previously been seen. This function is intended to be used where non-TeX processes may result in file deletion, for example if LuaTeX is in use, <code>os.remove()</code> may be used to delete a file part-way through a run.

12.2.2 Information about files and file contents

Functions in this section return information about files as `expl3 str` data, *except* that the non-expandable functions set their return *token list* to `\q_no_value` if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use `\str_if_eq:nnTF` rather than `\tl_if_eq:nnTF` and so on.

<code>\file_hex_dump:n</code> ☆	<code>\file_hex_dump:n {<file name>}</code>
<code>\file_hex_dump:V</code> ☆	<code>\file_hex_dump:nnn {<file name>} {<start index>} {<end index>}</code>
<code>\file_hex_dump:nnn</code> ☆	Searches for <code><file name></code> using the current TeX search path and the additional paths controlled by <code>\l_file_search_path_seq</code> . It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TeX behavior there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The <code>{<start index>}</code> and <code>{<end index>}</code> values work as described for <code>\str_range:nnn</code> .
<code>\file_hex_dump:Vnn</code> ☆	

<code>\file_get_hex_dump:nN</code>	<code>\file_get_hex_dump:nN {<file name>} <t1 var></code>
<code>\file_get_hex_dump:VN</code>	<code>\file_get_hex_dump:nnnN {<file name>} {<start index>} {<end index>} <t1 var></code>
<code>\file_get_hex_dump:nNTF</code>	Sets the <code><t1 var></code> to the result of applying <code>\file_hex_dump:n/\file_hex_dump:nnn</code> to the <code><file></code> . If the file is not found, the <code><t1 var></code> will be set to <code>\q_no_value</code> .
<code>\file_get_hex_dump:VNTF</code>	
<code>\file_get_hex_dump:nnnN</code>	
<code>\file_get_hex_dump:VnnN</code>	
<code>\file_get_hex_dump:nnnNTF</code>	
<code>\file_get_hex_dump:VnnNTF</code>	

<code>\file_md5_hash:n ☆</code>	<code>\file_md5_hash:n {<file name>}</code>
<code>\file_md5_hash:V ☆</code>	Searches for <code><file name></code> using the current T _E X search path and the additional paths controlled by <code>\l_file_search_path_seq</code> . It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most T _E X behavior there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

<code>\file_get_md5_hash:nN</code>	<code>\file_get_md5_hash:nN {<file name>} <t1 var></code>
<code>\file_get_md5_hash:VN</code>	Sets the <code><t1 var></code> to the result of applying <code>\file_md5_hash:n</code> to the <code><file></code> . If the file is not found, the <code><t1 var></code> will be set to <code>\q_no_value</code> .
<code>\file_get_md5_hash:nNTF</code>	
<code>\file_get_md5_hash:VNTF</code>	

<code>\file_size:n ☆</code>	<code>\file_size:n {<file name>}</code>
<code>\file_size:V ☆</code>	Searches for <code><file name></code> using the current T _E X search path and the additional paths controlled by <code>\l_file_search_path_seq</code> . It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

<code>\file_get_size:nN</code>	<code>\file_get_size:nN {<file name>} <t1 var></code>
<code>\file_get_size:VN</code>	Sets the <code><t1 var></code> to the result of applying <code>\file_size:n</code> to the <code><file></code> . If the file is not found, the <code><t1 var></code> will be set to <code>\q_no_value</code> .
<code>\file_get_size:nNTF</code>	
<code>\file_get_size:VNTF</code>	

<code>\file_timestamp:n ☆</code>	<code>\file_timestamp:n {<file name>}</code>
<code>\file_timestamp:V ☆</code>	Searches for <code><file name></code> using the current T _E X search path and the additional paths controlled by <code>\l_file_search_path_seq</code> . It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:<year><month><day><hour><minute><second><offset>, where the latter may be Z (UTC) or <plus-minus><hours>'<minutes>'. When the file is not found, the result of expansion is empty.

<code>\file_get_timestamp:nN</code>	<code>\file_get_timestamp:nN {<file name>} <t1 var></code>
<code>\file_get_timestamp:VN</code>	Sets the <code><t1 var></code> to the result of applying <code>\file_timestamp:n</code> to the <code><file></code> . If the file is not found, the <code><t1 var></code> will be set to <code>\q_no_value</code> .
<code>\file_get_timestamp:nNTF</code>	
<code>\file_get_timestamp:VNTF</code>	

<code>\file_compare_timestamp_p:nNn</code> <code>\file_compare_timestamp_p:(nNV VNn VNV)</code> <code>\file_compare_timestamp:nNnTF</code> <code>\file_compare_timestamp:(nNV VNn VNV)TF</code>	<code>* \file_compare_timestamp_p:nNn {<file-1>} <relation> {<file-2>}</code> <code>* \file_compare_timestamp_p:nNnTF {<file-1>} <relation> {<file-2>} {<true code>} {<false code>}</code> <code>* \file_compare_timestamp:nNnTF {<file-1>} <relation> {<file-2>} {<true code>} {<false code>}</code> <code>* \file_compare_timestamp:(nNV VNn VNV)TF</code>

Compares the file stamps on the two `<files>` as indicated by the `<relation>`, and inserts either the `<true code>` or `<false case>` as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
  % Code to regenerate derived file
}
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different.

<code>\file_get_full_name:nN</code> <code>\file_get_full_name:VN</code> <code>\file_get_full_name:nNnTF</code> <code>\file_get_full_name:VNnTF</code>	<code>\file_get_full_name:nN {<file name>} <tl var></code> <code>\file_get_full_name:nNnTF {<file name>} <tl var> {<true code>} {<false code>}</code>

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `<tl var>` the fully-qualified name of the file, i.e., the path and file name. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. In the non-branching version, the `<tl var>` will be set to `\q_no_value` in the case that the file does not exist.

<code>\file_full_name:n</code> ☆ <code>\file_full_name:V</code> ☆	<code>\file_full_name:n {<file name>}</code> <code>\file_full_name:V</code>

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found leaves the fully-qualified name of the file, i.e., the path and file name, in the input stream. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

<code>\file_parse_full_name:nNNN</code> <code>\file_parse_full_name:VNNN</code>	<code>\file_parse_full_name:nNNN {<full name>} <dir> <name> <ext></code> <code>\file_parse_full_name:VNNN</code>

Updated: 2020-06-24

Parses the `<full name>` and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The `<dir>`: everything up to the last / (path separator) in the `<file path>`. As with system PATH variables and related functions, the `<dir>` does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), `<dir>` is empty.
- The `<name>`: everything after the last / up to the last ., where both of those characters are optional. The `<name>` may contain multiple . characters. It is empty if `<full name>` consists only of a directory name.
- The `<ext>`: everything after the last . (including the dot). The `<ext>` is empty if there is no . after the last /.

Before parsing, the `<full name>` is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

<code>\file_parse_full_name:n</code> *	<code>\file_parse_full_name:n</code> { $\langle full\ name \rangle$ }
<code>\file_parse_full_name:V</code> *	Parses the $\langle full\ name \rangle$ as described for <code>\file_parse_full_name:nNNN</code> , and leaves
New: 2020-06-24	$\langle dir \rangle$, $\langle name \rangle$, and $\langle ext \rangle$ in the input stream, each inside a pair of braces.

<code>\file_parse_full_name_apply:nN</code> *	<code>\file_parse_full_name_apply:nN</code> { $\langle full\ name \rangle$ } $\langle function \rangle$
<code>\file_parse_full_name_apply:VN</code> *	
New: 2020-06-24	

Parses the $\langle full\ name \rangle$ as described for `\file_parse_full_name:nNNN`, and passes $\langle dir \rangle$, $\langle name \rangle$, and $\langle ext \rangle$ as arguments to $\langle function \rangle$, as an `n`-type argument each, in this order.

12.2.3 Accessing file contents

<code>\file_get:nnN</code>	<code>\file_get:nnN</code> { $\langle file\ name \rangle$ } { $\langle setup \rangle$ } $\langle tl\ var \rangle$
<code>\file_get:VnN</code>	<code>\file_get:nnNTF</code> { $\langle file\ name \rangle$ } { $\langle setup \rangle$ } $\langle tl\ var \rangle$ { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }
<code>\file_get:nnNTF</code>	Defines $\langle tl\ var \rangle$ to the contents of $\langle file\ name \rangle$. Category codes may need to be set appropriately via the $\langle setup \rangle$ argument. The non-branching version sets the $\langle tl\ var \rangle$ to <code>\q_no_value</code> if the file is not found. The branching version runs the $\langle true\ code \rangle$ after the assignment to $\langle tl\ var \rangle$ if the file is found, and $\langle false\ code \rangle$ otherwise. The file content will be tokenized using the current category code régime.
<code>\file_get:VnNTF</code>	

<code>\file_input:n</code>	<code>\file_input:n</code> { $\langle file\ name \rangle$ }
<code>\file_input:V</code>	Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional $\text{\LaTeX}$ source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

<code>\file_input_raw:n</code> *	<code>\file_input_raw:n</code> { $\langle file\ name \rangle$ }
<code>\file_input_raw:V</code> *	Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional $\text{\TeX}$ source. No data concerning the file is tracked. If the file is not found, no action is taken.
New: 2023-05-18	
Updated: 2025-05-26	

$\text{\TeX}$ hackers note: This function *requires* the availability of the `\input` primitive accepting braces (Lua $\text{\TeX}$ or other engines from $\text{\TeX}$ Live 2020 onwards.)

This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an `\halign`.

<code>\file_if_exist_input:n</code>	<code>\file_if_exist_input:n</code> { $\langle file\ name \rangle$ }
<code>\file_if_exist_input:V</code>	<code>\file_if_exist_input:nF</code> { $\langle file\ name \rangle$ } { $\langle false\ code \rangle$ }
<code>\file_if_exist_input:nF</code>	Searches for $\langle file\ name \rangle$ using the current $\text{\TeX}$ search path and the additional paths included in <code>\l_file_search_path_seq</code> . If found then reads in the file as additional $\text{\LaTeX}$ source as described for <code>\file_input:n</code> , otherwise inserts the $\langle false\ code \rangle$. Note that these functions do not raise an error if the file is not found, in contrast to <code>\file_input:n</code> .
<code>\file_if_exist_input:VF</code>	

`\file_input_stop:` `\file_input_stop:`

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

TeXhackers note: This function must be used on a line on its own: TeX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

`\file_show_list:` `\file_show_list:`
`\file_log_list:` `\file_log_list:`

These functions list all files loaded by L^AT_EX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Chapter 13

The `l3luatex` module LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

13.1 Breaking out to Lua

<code>\lua_now:n</code>	★	<code>\lua_now:n {⟨token list⟩}</code>
<code>\lua_now:e</code>	★	

The `⟨token list⟩` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the `⟨Lua input⟩` immediately, and in an expandable manner.

TeXhackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

<code>\lua_shipout_e:n</code>		<code>\lua_shipout:n {⟨token list⟩}</code>
<code>\lua_shipout:n</code>		

The `⟨token list⟩` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter when the current page is finalized (i.e., at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` during the page-building routine: no TeX expansion of the `⟨Lua input⟩` will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TeX in an e-type manner during the shipout operation.

TeXhackers note: At a TeX level, the `⟨Lua input⟩` is stored as a “whatsit”.

`\lua_escape:n` ★ `\lua_escape:n {⟨token list⟩}`

`\lua_escape:e` ★ Converts the `⟨token list⟩` such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

TeXhackers note: `\lua_escape:e` is a macro wrapper around `\luaescapestring:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

`\lua_load_module:n` `\lua_load_module:n {⟨Lua module name⟩}`

New: 2022-05-14 Loads a Lua module into the Lua interpreter.

`\lua_now:n` passes its `{⟨token list⟩}` argument to the Lua interpreter as a single line, with characters interpreted under the current catcode régime. These two facts mean that `\lua_now:n` rarely behaves as expected for larger pieces of code. Therefore, package authors should **not** write significant amounts of Lua code in the arguments to `\lua_now:n`. Instead, it is strongly recommended that they write the majority of their Lua code in a separate file, and then load it using `\lua_load_module:n`.

TeXhackers note: This is a wrapper around the Lua call `require '⟨module⟩'`.

13.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here.

`ltx.utils` Most public interfaces provided by the module are stored within the `ltx.utils` table.

`ltx.utils.filedump` `⟨dump⟩ = ltx.utils.filedump(⟨file⟩,⟨offset⟩,⟨length⟩)`

Returns the uppercase hexadecimal representation of the content of the `⟨file⟩` read as bytes. If the `⟨length⟩` is given, only this part of the file is returned; similarly, one may specify the `⟨offset⟩` from the start of the file. If the `⟨length⟩` is not given, the entire file is read starting at the `⟨offset⟩`.

`ltx.utils.filemd5sum` `⟨hash⟩ = ltx.utils.filemd5sum(⟨file⟩)`

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TeX behavior. If the `⟨file⟩` is not found, nothing is returned with *no error raised*.

`ltx.utils.filemoddate` `⟨date⟩ = ltx.utils.filemoddate(⟨file⟩)`

Returns the date/time of last modification of the `⟨file⟩` in the format

`D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minute⟩⟨second⟩⟨offset⟩`

where the latter may be Z (UTC) or `⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'`. If the `⟨file⟩` is not found, nothing is returned with *no error raised*.

`ltx.utils.filesize` `size = ltx.utils.filesize(<file>)`

Returns the size of the `<file>` in bytes. If the `<file>` is not found, nothing is returned with *no error raised*.

Chapter 14

The l3legacy module

Interfaces to legacy concepts

There are a small number of T_EX or L^AT_EX 2_ε concepts which are not used in expl3 code but which need to be manipulated when working as a L^AT_EX 2_ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

<code>\legacy_if_p:n</code>	★	<code>\legacy_if_p:n {<name>}</code>
<code>\legacy_if:nTF</code>	★	<code>\legacy_if:nTF {<name>} {<true code>} {<false code>}</code>

Tests if the L^AT_EX 2_ε/plain T_EX conditional (generated by `\newif`) is `true` or `false` and branches accordingly. The `<name>` of the conditional should *omit* the leading `if`.

<code>\legacy_if_set_true:n</code>	<code>\legacy_if_set_true:n {<name>}</code>
<code>\legacy_if_set_false:n</code>	<code>\legacy_if_set_false:n {<name>}</code>

<code>\legacy_if_gset_true:n</code>	Sets the L ^A T _E X 2 _ε /plain T _E X conditional <code>\if<name></code> (generated by <code>\newif</code>) to be <code>true</code> or <code>false</code> .
<code>\legacy_if_gset_false:n</code>	

New: 2021-05-10

<code>\legacy_if_set:nn</code>	<code>\legacy_if_set:nn {<name>} {<boolexpr>}</code>
<code>\legacy_if_gset:nn</code>	Sets the L ^A T _E X 2 _ε /plain T _E X conditional <code>\if<name></code> (generated by <code>\newif</code>) to the result of evaluating the <code><boolean expression></code> .

New: 2021-05-10

Part IV

Data types

Chapter 15

The `\l3tl` module

Token lists

$\text{\TeX}$ works with tokens, and $\text{\LaTeX3}$ therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “tl var” (`\tl var`), which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `\`, `{`, or `}` (assuming normal $\text{\TeX}$ category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, `\`, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initializing token list variables

```
\tl_new:N \tl_new:N <tl var>
```

```
\tl_new:c
```

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

<code>\tl_const:Nn</code>	<code>\tl_const:Nn <tl var> {<tokens>}</code>
<code>\tl_const:(NV Ne cn cV ce)</code>	Creates a new constant <code><tl var></code> or raises an error if the name is already taken. The value of the <code><tl var></code> is set globally to the <code><tokens></code> .
<code>\tl_clear:N</code>	<code>\tl_clear:N <tl var></code>
<code>\tl_clear:c</code>	
<code>\tl_gclear:N</code>	Clears all entries from the <code><tl var></code> .
<code>\tl_gclear:c</code>	
<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	
<code>\tl_gclear_new:N</code>	Ensures that the <code><tl var></code> exists globally by applying <code>\tl_new:N</code> if necessary, then applies <code>\tl_(g)clear:N</code> to leave the <code><tl var></code> empty.
<code>\tl_gclear_new:c</code>	
<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var_1> <tl var_2></code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of <code><tl var_1></code> equal to that of <code><tl var_2></code> .
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	
<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var_1> <tl var_2> <tl var_3></code>
<code>\tl_concat:ccc</code>	
<code>\tl_gconcat:NNN</code>	Concatenates the content of <code><tl var_2></code> and <code><tl var_3></code> together and saves the result in <code><tl var_1></code> . The <code><tl var_2></code> is placed at the left side of the new token list.
<code>\tl_gconcat:ccc</code>	
<code>\tl_if_exist_p:N *</code>	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c *</code>	<code>\tl_if_exist:NTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_exist:N$\overline{TF}$ *</code>	
<code>\tl_if_exist:c$\overline{TF}$ *</code>	Tests whether the <code><tl var></code> is currently defined. This does not check that the <code><tl var></code> really is a token list variable.

15.2 Adding data to token list variables

<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {<tokens>}</code>
<code>\tl_set:(NV Nv No Ne Nf cn cV cv co ce cf)</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Ne Nf cn cV cv co ce cf)</code>	
	Sets <code><tl var></code> to contain <code><tokens></code> , removing any previous content from the variable.
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {<tokens>}</code>
<code>\tl_put_left:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV Nv Ne No cn cV cv ce co)</code>	
	Appends <code><tokens></code> to the left side of the current content of <code><tl var></code> .

<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {<tokens>}</code>
<code>\tl_put_right:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\tl_gput_right:Nn</code>	
<code>\tl_gput_right:(NV Nv Ne No cn cV cv ce co)</code>	

Appends `<tokens>` to the right side of the current content of `<tl var>`.

15.3 Token list conditionals

<code>\tl_if_blank_p:n</code>	<code>* \tl_if_blank_p:n {<token list>}</code>
<code>\tl_if_blank_p:(e V o)</code>	<code>* \tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_blank:nTF</code>	<code>* Tests if the <token list> consists only of blank spaces (i.e., contains no item). The test</code>
<code>\tl_if_blank:(e V o)TF</code>	<code>* is true if <token list> is zero or more explicit space characters (explicit tokens with</code>

`character code 32 and category code 10), and is false otherwise.`

<code>\tl_if_empty_p:N</code>	<code>* \tl_if_empty_p:N <tl var></code>
<code>\tl_if_empty_p:c</code>	<code>* \tl_if_empty:nTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	<code>* Tests if the <tl var> is entirely empty (i.e., contains no tokens at all).</code>
<code>\tl_if_empty:cTF</code>	

<code>\tl_if_empty_p:n</code>	<code>* \tl_if_empty_p:n {<token list>}</code>
<code>\tl_if_empty_p:(V o e)</code>	<code>* \tl_if_empty:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	<code>* Tests if the <token list> is entirely empty (i.e., contains no tokens at all).</code>
<code>\tl_if_empty:(V o e)TF</code>	

<code>\tl_if_eq_p:NN</code>	<code>* \tl_if_eq_p:NN <tl var₁> <tl var₂></code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	<code>* \tl_if_eq:nNTF <tl var₁> <tl var₂> {<true code>} {<false code>}</code>
<code>\tl_if_eq:nNTF</code>	<code>* Compares the content of <tl var₁> and <tl var₂> and is logically true if the two contain</code>
<code>\tl_if_eq:(Nc cN cc)TF</code>	<code>* the same list of tokens (i.e., identical in both the list of characters they contain and the</code>

`category codes of those characters). Thus for example`

```

\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Ne \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:nNTF \l_tmpa_tl \l_tmpb_tl { true } { false }

```

yields false. See also `\str_if_eq:nNTF` for a comparison that ignores category codes.

<code>\tl_if_eq:nNTF</code>	<code>\tl_if_eq:nNTF <tl var₁> {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_eq:cnTF</code>	<code>Tests if the <tl var₁> and the <token list₂> contain the same list of tokens, both in</code>
New: 2020-07-14	<code>respect of character codes and category codes. This conditional is not expandable: see</code>
	<code>\tl_if_eq:nNTF for an expandable version when both token lists are stored in variables,</code>
	<code>or \str_if_eq:nNTF if category codes are not important.</code>

<code>\tl_if_eq:nnTF</code>	<code>\tl_if_eq:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_eq:(nV ne Vn en ee)TF</code>	<code>Tests if <token list₁> and <token list₂> contain the same list of tokens, both in respect</code>
	<code>of character codes and category codes. This conditional is not expandable: see \tl_if_</code>
	<code>eq:nNTF for an expandable version when token lists are stored in variables, or \str_</code>
	<code>if_eq:nnTF if category codes are not important.</code>

$\backslash\text{tl\_if\_in:NnTF}$ $\backslash\text{tl\_if\_in:}\langle\text{NV No cn cV co}\rangle\text{TF}$	$\backslash\text{tl\_if\_in:NnTF} \langle\text{tl var}\rangle \{\langle\text{token list}\rangle\} \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if the $\langle\text{token list}\rangle$ is found in the content of the $\langle\text{tl var}\rangle$. The $\langle\text{token list}\rangle$ cannot contain the tokens $\{, \}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

$\backslash\text{tl\_if\_in:nnTF}$ $\backslash\text{tl\_if\_in:}\langle\text{Vn VV on oo nV no}\rangle\text{TF}$	$\backslash\text{tl\_if\_in:nnTF} \{\langle\text{token list}_1\rangle\} \{\langle\text{token list}_2\rangle\} \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if $\langle\text{token list}_2\rangle$ is found inside $\langle\text{token list}_1\rangle$. The $\langle\text{token list}_2\rangle$ cannot contain the tokens $\{, \}$ or $\#$ (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does <i>not</i> enter brace (category code 1/2) groups.

$\backslash\text{tl\_if\_novalue\_p:n} \star$ $\backslash\text{tl\_if\_novalue:nTF} \star$	$\backslash\text{tl\_if\_novalue\_p:n} \{\langle\text{token list}\rangle\}$ $\backslash\text{tl\_if\_novalue:nTF} \{\langle\text{token list}\rangle\} \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if the $\langle\text{token list}\rangle$ and the special $\backslash\text{c\_novalue\_tl}$ marker contain the same list of tokens, both in respect of character codes and category codes. This means that $\backslash\text{exp\_args:No} \backslash\text{tl\_if\_novalue:nTF} \{ \backslash\text{c\_novalue\_tl} \}$ is logically true but $\backslash\text{tl\_if\_novalue:nTF} \{ \backslash\text{c\_novalue\_tl} \}$ is logically false. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

$\backslash\text{tl\_if\_single\_p:N} \star$ $\backslash\text{tl\_if\_single\_p:c} \star$ $\backslash\text{tl\_if\_single:NTF} \star$ $\backslash\text{tl\_if\_single:cTF} \star$	$\backslash\text{tl\_if\_single\_p:N} \langle\text{tl var}\rangle$ $\backslash\text{tl\_if\_single:NTF} \langle\text{tl var}\rangle \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if the content of the $\langle\text{tl var}\rangle$ consists of a single $\langle\text{item}\rangle$, i.e., is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to $\backslash\text{tl\_count:N}$.

$\backslash\text{tl\_if\_single\_p:n} \star$ $\backslash\text{tl\_if\_single:nTF} \star$	$\backslash\text{tl\_if\_single\_p:n} \{\langle\text{token list}\rangle\}$ $\backslash\text{tl\_if\_single:nTF} \{\langle\text{token list}\rangle\} \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if the $\langle\text{token list}\rangle$ has exactly one $\langle\text{item}\rangle$, i.e., is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to $\backslash\text{tl\_count:n}$.

$\backslash\text{tl\_if\_single\_token\_p:n} \star$ $\backslash\text{tl\_if\_single\_token:nTF} \star$	$\backslash\text{tl\_if\_single\_token\_p:n} \{\langle\text{token list}\rangle\}$ $\backslash\text{tl\_if\_single\_token:nTF} \{\langle\text{token list}\rangle\} \{\langle\text{true code}\rangle\} \{\langle\text{false code}\rangle\}$ Tests if the token list consists of exactly one token, i.e., is either a single space character or a single normal token. Token groups ($\{...\}$) are not single tokens.

```

\tl_if_regex_match:nnTF \tl_if_regex_match:nnTF {<token list>} {<regex>} {<true code>} {<false code>}
\tl_if_regex_match:VnTF \tl_if_regex_match:nNTF {<token list>} <regex var> {<true code>} {<false code>}
\tl_if_regex_match:nNTF
\tl_if_regex_match:VNTF

```

Tests whether the *<regular expression>* matches any part of the *<token list>*. For instance,

New: 2024-12-08

```

\tl_if_regex_match:nnTF { abecdcx } { b [cde]* } { TRUE } { FALSE }
\tl_if_regex_match:nnTF { example } { [b-dq-w] } { TRUE } { FALSE }

```

leaves TRUE then FALSE in the input stream. Theses are alternative names for `\regex_if_match:nnTF` and friends, with arguments re-ordered for *<token list>* testing; see `l3regex` chapter for more details of the *<regex>* format.

15.3.1 Testing the first token

```

\tl_if_head_eq_catcode_p:nN      * \tl_if_head_eq_catcode_p:nN {<token list>} <test token>
\tl_if_head_eq_catcode_p:(VN|eN|oN) * \tl_if_head_eq_catcode:nNTF {<token list>} <test token>
\tl_if_head_eq_catcode:nNTF      * {<true code>} {<false code>}
\tl_if_head_eq_catcode:(VN|eN|oN)TF *

```

Tests if the first *<token>* in the *<token list>* has the same category code as the *<test token>*. In the case where the *<token list>* is empty, the test is always false.

```

\tl_if_head_eq_charcode_p:nN      * \tl_if_head_eq_charcode_p:nN {<token list>} <test token>
\tl_if_head_eq_charcode_p:(VN|eN|fN) * \tl_if_head_eq_charcode:nNTF {<token list>} <test token>
\tl_if_head_eq_charcode:nNTF      * {<true code>} {<false code>}
\tl_if_head_eq_charcode:(VN|eN|fN)TF *

```

Tests if the first *<token>* in the *<token list>* has the same character code as the *<test token>*. In the case where the *<token list>* is empty, the test is always false.

```

\tl_if_head_eq_meaning_p:nN      * \tl_if_head_eq_meaning_p:nN {<token list>} <test token>
\tl_if_head_eq_meaning_p:(VN|eN) * \tl_if_head_eq_meaning:nNTF {<token list>} <test token>
\tl_if_head_eq_meaning:nNTF      * {<true code>} {<false code>}
\tl_if_head_eq_meaning:(VN|eN)TF *

```

Tests if the first *<token>* in the *<token list>* has the same meaning as the *<test token>*. In the case where *<token list>* is empty, the test is always false.

```

\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {<token list>}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {<token list>} {<true code>} {<false code>}

```

Tests if the first *<token>* in the *<token list>* is an explicit begin-group character (with category code 1 and any character code), in other words, if the *<token list>* starts with a brace group. In particular, the test is false if the *<token list>* starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```

\if_tl_head_is_N_type_p:n * \if_tl_head_is_N_type_p:n {<token list>}
\if_tl_head_is_N_type:nTF * \if_tl_head_is_N_type:nTF {<token list>} {<true code>} {<false code>}

```

Tests if the first *<token>* in the *<token list>* is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.

```

\if_tl_head_is_space_p:n * \if_tl_head_is_space_p:n {<token list>}
\if_tl_head_is_space:nTF * \if_tl_head_is_space:nTF {<token list>} {<true code>} {<false code>}

```

Tests if the first *<token>* in the *<token list>* is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

```

\tl_to_str:n * \tl_to_str:n {<token list>}
\tl_to_str:(o|V|v|e) *

```

Converts the *<token list>* to a *<string>*, leaving the resulting character tokens in the input stream. A *<string>* is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). The base function requires only a single expansion. Its argument *must* be braced.

T_EXhackers note: This is the ε -T_EX primitive `\detokenize`. Converting a *<token list>* to a *<string>* yields a concatenation of the string representations of every token in the *<token list>*. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

<code>\tl_to_str:N</code>	<code>*</code>	<code>\tl_to_str:N <tl var></code>
<code>\tl_to_str:c</code>	<code>*</code>	Converts the content of the <code><tl var></code> into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This <code><string></code> is then left in the input stream. For low-level details, see the notes given for <code>\tl_to_str:n</code> .

<code>\tl_use:N</code>	<code>*</code>	<code>\tl_use:N <tl var></code>
<code>\tl_use:c</code>	<code>*</code>	Recovers the content of a <code><tl var></code> and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a <code><tl var></code> directly without an accessor function.

15.4.2 Counting and reversing token lists

<code>\tl_count:n</code>	<code>*</code>	<code>\tl_count:n {(token list)}</code>
<code>\tl_count:(V v e o)</code>	<code>*</code>	Counts the number of <code><items></code> in the <code><token list></code> and leaves this information in the input stream. Unbraced tokens count as one element as do each token group <code>{...}</code> . This process ignores any unprotected spaces within the <code><token list></code> . See also <code>\tl_count:N</code> . This function requires three expansions, giving an <code><integer denotation></code> .

<code>\tl_count:N</code>	<code>*</code>	<code>\tl_count:N <tl var></code>
<code>\tl_count:c</code>	<code>*</code>	Counts the number of <code><items></code> in the <code><tl var></code> and leaves this information in the input stream. Unbraced tokens count as one element as do each token group <code>{...}</code> . This process ignores any unprotected spaces within the <code><tl var></code> . See also <code>\tl_count:n</code> . This function requires three expansions, giving an <code><integer denotation></code> .

<code>\tl_count_tokens:n</code>	<code>*</code>	<code>\tl_count_tokens:n {(token list)}</code>
		Counts the number of $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ tokens in the <code><token list></code> and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of <code>a~{bc}</code> is 6.

<code>\tl_reverse:n</code>	<code>*</code>	<code>\tl_reverse:n {(token list)}</code>
<code>\tl_reverse:(V o f e)</code>	<code>*</code>	Reverses the order of the <code><items></code> in the <code><token list></code> , so that <code><item₁><item₂><item₃>...<item_n></code> becomes <code><item_n>...<item₃><item₂><item₁></code> . This process preserves unprotected space within the <code><token list></code> . Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider <code>\tl_reverse_items:n</code> . See also <code>\tl_reverse:N</code> .

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\tl_reverse:N</code>	<code>\tl_reverse:N <tl var></code>
<code>\tl_reverse:c</code>	
<code>\tl_greverse:N</code>	
<code>\tl_greverse:c</code>	Sets the <code><tl var></code> to contain the result of reversing the order of its <code><items></code> , so that <code><item₁><item₂><item₃>...<item_n></code> becomes <code><item_n>...<item₃><item₂><item₁></code> . This process preserves unprotected spaces within the <code><tl var></code> . Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and <code>\tl_reverse:V</code> . See also <code>\tl_reverse_items:n</code> for improved performance.

<code>\tl_reverse_items:n</code>	<code>\tl_reverse_items:n {(token list)}</code>

Reverses the order of the `<items>` in the `<token list>`, so that `<item1><item2><item3>...<itemn>` becomes `{<itemn>} ... {<item3>}{<item2>}{<item1>}`. This process removes any unprotected space within the `<token list>`. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an **e**-type or **x**-type argument expansion.

<code>\tl_trim_spaces:n</code>	<code>\tl_trim_spaces:n {(token list)}</code>
<code>\tl_trim_spaces:(V v e o)</code>	

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the `<token list>` and leaves the result in the input stream.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an **e**-type or **x**-type argument expansion.

<code>\tl_trim_left_spaces:n</code>	<code>\tl_trim_left_spaces:n {(token list)}</code>
<code>\tl_trim_left_spaces:(V v e o)</code>	
<code>\tl_trim_right_spaces:n</code>	
<code>\tl_trim_right_spaces:(V v e o)</code>	

New: 2025-02-02

Analogue of `\tl_trim_spaces:n` which removes any leading *or* trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the `<token list>` and leaves the result in the input stream.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an **e**-type or **x**-type argument expansion.

<code>\tl_trim_spaces_apply:nN</code>	<code>\tl_trim_spaces_apply:nN {(token list)} <function></code>
<code>\tl_trim_spaces_apply:oN</code>	

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the `<token list>` and passes the result to the `<function>` as an **n**-type argument.

<code>\tl_trim_left_spaces_apply:nN</code>	<code>*</code>	<code>\tl_trim_left_spaces_apply:nN {⟨token list⟩} ⟨function⟩</code>
<code>\tl_trim_left_spaces_apply:oN</code>	<code>*</code>	
<code>\tl_trim_right_spaces_apply:nN</code>	<code>*</code>	
<code>\tl_trim_right_spaces_apply:oN</code>	<code>*</code>	

New: 2025-02-02

Analogue of `\tl_trim_spaces_apply:nN` which removes any leading *or* trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the `⟨token list⟩` and passes the result to the `⟨function⟩` as an `n`-type argument.

<code>\tl_trim_spaces:N</code>	<code>\tl_trim_spaces:N ⟨tl var⟩</code>
<code>\tl_trim_spaces:c</code>	
<code>\tl_gtrim_spaces:N</code>	Sets the <code>⟨tl var⟩</code> to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.
<code>\tl_gtrim_spaces:c</code>	

<code>\tl_trim_left_spaces:N</code>	<code>\tl_trim_left_spaces:N ⟨tl var⟩</code>
<code>\tl_trim_left_spaces:c</code>	
<code>\tl_trim_right_spaces:N</code>	
<code>\tl_trim_right_spaces:c</code>	Analogue of <code>\tl_trim_spaces:N</code> which sets the <code>⟨tl var⟩</code> to contain the result of removing any leading <i>or</i> trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.
<code>\tl_gtrim_left_spaces:N</code>	
<code>\tl_gtrim_left_spaces:c</code>	
<code>\tl_gtrim_right_spaces:N</code>	
<code>\tl_gtrim_right_spaces:c</code>	

New: 2025-02-02

15.4.3 Viewing token lists

<code>\tl_show:N</code>	<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>	
Updated: 2021-04-29	Displays the content of the <code>⟨tl var⟩</code> on the terminal.

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\tl_show:n</code>	<code>\tl_show:n {⟨token list⟩}</code>
<code>\tl_show:e</code>	
	Displays the <code>⟨token list⟩</code> on the terminal.

TeXhackers note: This is similar to the ϵ -TeX primitive `\showtokens`, wrapped to a fixed number of characters per line.

<code>\tl_log:N</code>	<code>\tl_log:N ⟨tl var⟩</code>
<code>\tl_log:c</code>	
Updated: 2021-04-29	Writes the content of the <code>⟨tl var⟩</code> in the log file. See also <code>\tl_show:N</code> which displays the result in the terminal.

<code>\tl_log:n</code>	<code>\tl_log:n {⟨token list⟩}</code>
<code>\tl_log:(e x)</code>	
	Writes the <code>⟨token list⟩</code> in the log file. See also <code>\tl_show:n</code> which displays the result in the terminal.

15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, i.e., any local assignments made by the $\langle function \rangle$ or $\langle code \rangle$ discussed below remain in effect after the loop.

$\backslash\text{tl\_map\_function:Nn}$ ☆	$\backslash\text{tl\_map\_function:Nn}$ $\langle\text{tl var}\rangle$ $\langle function \rangle$
$\backslash\text{tl\_map\_function:cN}$ ☆	Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle\text{tl var}\rangle$. The $\langle function \rangle$ receives one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also $\backslash\text{tl\_map\_function:nN}$.

$\backslash\text{tl\_map\_function:nN}$ ☆	$\backslash\text{tl\_map\_function:nN}$ $\{\langle token list \rangle\}$ $\langle function \rangle$
$\backslash\text{tl\_map\_function:eN}$ ☆	Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle token list \rangle$, The $\langle function \rangle$ receives one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also $\backslash\text{tl\_map\_function:NN}$.

$\backslash\text{tl\_map\_inline:Nn}$	$\backslash\text{tl\_map\_inline:Nn}$ $\langle\text{tl var}\rangle$ $\{\langle inline function \rangle\}$
$\backslash\text{tl\_map\_inline:cn}$	Applies the $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle\text{tl var}\rangle$. The $\langle inline function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. See also $\backslash\text{tl\_map\_function:NN}$.

$\backslash\text{tl\_map\_inline:nn}$	$\backslash\text{tl\_map\_inline:nn}$ $\{\langle token list \rangle\}$ $\{\langle inline function \rangle\}$
	Applies the $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle token list \rangle$. The $\langle inline function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. See also $\backslash\text{tl\_map\_function:nN}$.

$\backslash\text{tl\_map\_tokens:Nn}$ ☆	$\backslash\text{tl\_map\_tokens:Nn}$ $\langle\text{tl var}\rangle$ $\{\langle code \rangle\}$
$\backslash\text{tl\_map\_tokens:cn}$ ☆	$\backslash\text{tl\_map\_tokens:nn}$ $\{\langle token list \rangle\}$ $\{\langle code \rangle\}$
$\backslash\text{tl\_map\_tokens:nn}$ ☆	Analogue of $\backslash\text{tl\_map\_function:NN}$ which maps several tokens instead of a single function. The $\langle code \rangle$ receives each $\langle item \rangle$ in the $\langle\text{tl var}\rangle$ or in the $\langle token list \rangle$ as a trailing brace group. For instance,

$\backslash\text{tl\_map\_tokens:Nn}$ $\backslash\text{my\_tl}$ { $\backslash\text{prg\_replicate:nn}$ { 2 } }

expands to twice each $\langle item \rangle$ in the $\langle\text{tl var}\rangle$: for each $\langle item \rangle$ in $\backslash\text{my\_tl}$ the function $\backslash\text{prg\_replicate:nn}$ receives 2 and $\langle item \rangle$ as its two arguments. The function $\backslash\text{tl\_map\_inline:Nn}$ is typically faster but is not expandable.

$\backslash\text{tl\_map\_variable:NNn}$	$\backslash\text{tl\_map\_variable:NNn}$ $\langle\text{tl var}\rangle$ $\langle variable \rangle$ $\{\langle code \rangle\}$
$\backslash\text{tl\_map\_variable:cNn}$	Stores each $\langle item \rangle$ of the $\langle\text{tl var}\rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. Its value after the loop is the last $\langle item \rangle$ in the $\langle\text{tl var}\rangle$, or its original value if the $\langle\text{tl var}\rangle$ is blank. See also $\backslash\text{tl\_map\_inline:Nn}$.

`\tl_map_variable:nNn` `\tl_map_variable:nNn {<token list>} <variable> {<code>}`

Stores each *<item>* of the *<token list>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<tl var>*, or its original value if the *<tl var>* is blank. See also `\tl_map_inline:nn`.

`\tl_map_break: ☆` `\tl_map_break:`

Used to terminate a `\tl_map...` function before all entries in the *<token list>* have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
  % Do something useful
}
```

See also `\tl_map_break:n`. Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\tl_map_break:n ☆` `\tl_map_break:n {<code>}`

Used to terminate a `\tl_map...` function before all entries in the *<token list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\tl_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

<code>\tl_head:N</code>	★	<code>\tl_head:n {⟨token list⟩}</code>
<code>\tl_head:n</code>	★	
<code>\tl_head:(V v f e)</code>	★	Leaves in the input stream the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , discarding the rest of the <i>⟨token list⟩</i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

`\tl_head:n { ~ { ~ ab } c }`

yields `␣ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *e*-type or *x*-type argument expansion.

<code>\tl_head:w</code>	★	<code>\tl_head:w ⟨token list⟩ { } \q_stop</code>

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *⟨token list⟩* (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an *o*-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:N</code>	★	<code>\tl_tail:n {⟨token list⟩}</code>
<code>\tl_tail:n</code>	★	
<code>\tl_tail:(V v f e)</code>	★	Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , and leaves the remaining tokens in the input stream. Thus for example

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

both leave `␣{bc}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *e*-type or *x*-type argument expansion.

If you wish to handle token lists where the first token may be a space, and this

needs to be treated as the head/tail, this can be accomplished using `\tl_if_head_is_space:nTF`, for example

```
\exp_last_unbraced:NNo
\cs_new:Npn \__mypkg_gobble_space:w \c_space_tl { }
\cs_new:Npn \mypkg_tl_head_keep_space:n #1
{
  \tl_if_head_is_space:nTF {#1}
  { ~ }
  { \tl_head:n {#1} }
}
\cs_new:Npn \mypkg_tl_tail_keep_space:n #1
{
  \tl_if_head_is_space:nTF {#1}
  { \exp_not:o { \__mypkg_gobble_space:w #1 } }
  { \tl_tail:n {#1} }
}
```

15.5.3 Items and ranges in token lists

<code>\tl_item:nn</code> *	<code>\tl_item:nn {<token list>} {<integer expression>}</code>
<code>\tl_item:Nn</code> *	Indexing items in the <code><token list></code> from 1 on the left, this function evaluates the <code><integer expression></code> and leaves the appropriate item from the <code><token list></code> in the input stream. If the <code><integer expression></code> is negative, indexing occurs from the right of the token list, starting at <code>-1</code> for the right-most item. If the index is out of bounds, then the function expands to nothing.
<code>\tl_item:cn</code> *	

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\tl_rand_item:N</code> *	<code>\tl_rand_item:N <tl var></code>
<code>\tl_rand_item:c</code> *	<code>\tl_rand_item:n {<token list>}</code>
<code>\tl_rand_item:n</code> *	Selects a pseudo-random item of the <code><token list></code> . If the <code><token list></code> is blank, the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

```

\tl_range:Nnn * \tl_range:Nnn <tl var> {\start index} {\end index}
\tl_range:nnn * \tl_range:nnn {\token list} {\start index} {\end index}

```

Leaves in the input stream the items from the $\langle \textit{start index} \rangle$ to the $\langle \textit{end index} \rangle$ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here $\langle \textit{start index} \rangle$ and $\langle \textit{end index} \rangle$ should be $\langle \textit{integer expressions} \rangle$. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with).

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```

\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }

```

Here are some more interesting examples. The calls

```

\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }

```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```

\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }

```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```

\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }

```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list $\langle \textit{tl} \rangle$, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle \textit{item} \rangle$ does not expand further when appearing in an *e*-type or *x*-type argument expansion.

15.5.4 Formatting token lists

<code>\tl_format:Nn</code> *	<code>\tl_format:Nn <tl var> {<format specification>}</code>
<code>\tl_format:cn</code> *	<code>\tl_format:nn {<token list>} {<format specification>}</code>
<code>\tl_format:nn</code> *	Converts the <code><tl var></code> or <code><token list></code> to a string according to the <code><format specification></code> .
New: 2025-06-09	The <code><style></code> , if present, must be <code>s</code> . If <code><precision></code> is given, all characters of the string representation of the <code><token list></code> beyond the first <code><precision></code> characters are discarded. The details of the <code><format specification></code> are described in Section 19.1.

15.5.5 Sorting token lists

<code>\tl_sort:Nn</code>	<code>\tl_sort:Nn <tl var> {<comparison code>}</code>
<code>\tl_sort:cn</code>	
<code>\tl_gsort:Nn</code>	Sorts the items in the <code><tl var></code> according to the <code><comparison code></code> , and assigns the result to <code><tl var></code> . The details of sorting comparison are described in Section 6.1.
<code>\tl_gsort:cn</code>	
<code>\tl_sort:nN</code> *	<code>\tl_sort:nN {<token list>} <conditional></code>
	Sorts the items in the <code><token list></code> , using the <code><conditional></code> to compare items, and leaves the result in the input stream. The <code><conditional></code> should have signature <code>:nnTF</code> , and return <code>true</code> if the two items being compared should be left in the same order, and <code>false</code> if the items should be swapped. The details of sorting comparison are described in Section 6.1.

T_EXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

15.6 Manipulating tokens in token lists

15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

<code>\tl_replace_once:Nnn</code>	<code>\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_once:(NVn NnV Nen Nne Nee cnn cVn cnV cen cne cee)</code>	
<code>\tl_greplace_once:Nnn</code>	
<code>\tl_greplace_once:(NVn NnV Nen Nne Nee cnn cVn cnV cen cne cee)</code>	

Replaces the first (leftmost) occurrence of `<old tokens>` in the `<tl var>` with `<new tokens>`. `<Old tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_replace_all:Nnn</code>	<code>\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_all:(NVn NnV Nen Nne Nee cnn cVn cnV cen cne cee)</code>	
<code>\tl_greplace_all:Nnn</code>	
<code>\tl_greplace_all:(NVn NnV Nen Nne Nee cnn cVn cnV cen cne cee)</code>	

Replaces all occurrences of `<old tokens>` in the `<tl var>` with `<new tokens>`. `<Old tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern `<old tokens>` may remain after the replacement (see `\tl_remove_all:Nn` for an example).

<code>\tl_regex_replace_once:Nnn</code>	<code>\tl_regex_replace_once:Nnn <tl var> {<regex>} {<replacement>}</code>
<code>\tl_regex_replace_once:cnn</code>	<code>\tl_regex_replace_once:NNn <tl var> <regex var> {<replacement>}</code>
<code>\tl_regex_replace_once:NNn</code>	
<code>\tl_regex_replace_once:cNn</code>	
<code>\tl_regex_greplace_once:Nnn</code>	
<code>\tl_regex_greplace_once:cnn</code>	
<code>\tl_regex_greplace_once:NNn</code>	
<code>\tl_regex_greplace_once:cNn</code>	

New: 2024-12-08

Searches for the `<regular expression>` in the contents of the `<tl var>` and replaces the first match with the `<replacement>`. In the `<replacement>`, `\0` represents the full match, `\1` represents the contents of the first capturing group, `\2` of the second, etc. Theses are alternative names for `\regex_replace_once:nnN` and friends, with arguments re-ordered for `<tl var>` setting; See `l3regex` chapter for more details of the `<regex>` format.

<code>\tl_regex_replace_all:Nnn</code>	<code>\tl_regex_replace_all:Nnn <tl var> {<regex>} {<replacement>}</code>
<code>\tl_regex_replace_all:cnn</code>	<code>\tl_regex_replace_all:NNn <tl var> <regex var> {<replacement>}</code>
<code>\tl_regex_replace_all:NNn</code>	
<code>\tl_regex_replace_all:cNn</code>	
<code>\tl_regex_greplace_all:Nnn</code>	
<code>\tl_regex_greplace_all:cnn</code>	
<code>\tl_regex_greplace_all:NNn</code>	
<code>\tl_regex_greplace_all:cNn</code>	

New: 2024-12-08

Replaces all occurrences of the `<regular expression>` in the contents of the `<tl var>` by the `<replacement>`, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, etc. Every match is treated independently, and matches cannot overlap. Theses are alternative names for `\regex_replace_all:nnN` and friends, with arguments re-ordered for `<tl var>` setting; see `l3regex` chapter for more details of the `<regex>` format.

<code>\tl_remove_once:Nn</code>	<code>\tl_remove_once:Nn <tl var> {<tokens>}</code>
<code>\tl_remove_once:(NV Ne cn cV ce)</code>	
<code>\tl_gremove_once:Nn</code>	
<code>\tl_gremove_once:(NV Ne cn cV ce)</code>	

Removes the first (leftmost) occurrence of `<tokens>` from the `<tl var>`. The `<tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_remove_all:Nn</code>	<code>\tl_remove_all:Nn <tl var> {<tokens>}</code>
<code>\tl_remove_all:(NV Ne cn cV ce)</code>	
<code>\tl_gremove_all:Nn</code>	
<code>\tl_gremove_all:(NV Ne cn cV ce)</code>	

Removes all occurrences of `<tokens>` from the `<tl var>`. The `<tokens>` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern `<tokens>` may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply T_EX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

<code>\tl_set_rescan:Nnn</code>	<code>\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}</code>
<code>\tl_set_rescan:(NnV Nne Nno cnn cnV cne cno)</code>	
<code>\tl_gset_rescan:Nnn</code>	
<code>\tl_gset_rescan:(NnV Nne Nno cnn cnV cne cno)</code>	

Sets `<tl var>` to contain `<tokens>`, applying the category code régime specified in the `<setup>` before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the `<setup>` are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the `<tl var>` to contain material with category codes other than those that apply when `<tokens>` are absorbed. The `<setup>` is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_rescan:nn`.

T_EXhackers note: The `<tokens>` are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user `<setup>`), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

`\tl_rescan:nn` `\tl_rescan:nn {<setup>} {<tokens>}`

`\tl_rescan:nV`

Rescans `<tokens>` applying the category code régime specified in the `<setup>`, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the `<setup>` are those in force at the point of use of `\tl_rescan:nn`.) The `<setup>` is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the `<tokens>` argument of `\tl_rescan:nn`.

TeXhackers note: The `<tokens>` are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user `<setup>`), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the `\scantokens` ε -TeX primitive, `\tl_rescan:nn` tokenizes the whole string in the same category code régime rather than one token at a time, so that directives such as `\verb` that rely on changing category codes will not function properly.

`\tl_retokenize:n` `\tl_retokenize:n {<tokens>}`

`\tl_retokenize:V`

New: 2025-07-08

Retokenizes the `<tokens>` by applying the current category code régime. In contrast to `\tl_rescan:nn`, this function executes the `<tokens>` as the category codes are reassigned to the tokens. As such, any category code changes within the `<tokens>` will apply to later content. The `tokens` are always treated as ending with a space. Within the `<tokens>`, the character `^^J` should be used to represent line breaks.

TeXhackers note: This is a thin wrapper around the `\scantokens` primitive. In addition to the primitive behavior, it detokenizes the input and resets the value of `\endlinechar` to 13 (i.e. `^^M`) and `\newlinechar` to 10 (i.e. `^^J`).

15.7 Constant token lists

`\c_empty_tl` Constant that is always empty.

`\c_novalue_tl`

A marker for the absence of an argument. This constant `tl` can safely be typeset (*cf.* `\q_nil`), with the result being `-NoValue-`. It is important to note that `\c_novalue_tl` is constructed such that it will *not* match the simple text input `-NoValue-`, i.e. that

`\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }`

is logically `false`. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

<code>\c_space_tl</code>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.

15.8 Scratch token lists

<code>\l_tmpa_tl</code> <code>\l_tmpb_tl</code>	Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code>	Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 16

The l3tl-build module

Piecewise tl constructions

16.1 Constructing $\langle tl\ var \rangle$ by accumulation

When creating a $\langle tl\ var \rangle$ by accumulation of many tokens, the performance available using a combination of `\tl_set:Nn` and `\tl_put_right:Nn` or similar begins to become an issue. To address this, a set of functions are available to “build” a $\langle tl\ var \rangle$. The performance of this approach is much more efficient than the standard `\tl_put_right:Nn`, but the constructed token list cannot be accessed during construction other than by methods provided in this section.

Whilst the exact performance difference is dependent on the size of each added block of tokens and the total number of blocks, in general, the `\tl_build_(g)put...` functions will out-perform the basic `\tl_(g)put...` equivalent if more than 100 non-empty addition operations occur. See <https://github.com/latex3/latex3/issues/1393#issuecomment-1880164756> for a more detailed analysis.

<code>\tl_build_begin:N</code>	<code>\tl_build_begin:N $\langle tl\ var \rangle$</code>

<code>\tl_build_gbegin:N</code>	

Clears the $\langle tl\ var \rangle$ and sets it up to support other `\tl_build_...` functions. Until `\tl_build_end:N  $\langle tl\ var \rangle$` or `\tl_build_gend:N  $\langle tl\ var \rangle$` is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global $\langle tl\ var \rangle$ respectively.

<code>\tl_build_put_left:Nn</code>	<code>\tl_build_put_left:Nn $\langle tl\ var \rangle$ $\{ \langle tokens \rangle \}$</code>

<code>\tl_build_put_left:Ne</code>	<code>\tl_build_put_right:Nn $\langle tl\ var \rangle$ $\{ \langle tokens \rangle \}$</code>

<code>\tl_build_gput_left:Nn</code>	

<code>\tl_build_gput_left:Ne</code>	

<code>\tl_build_put_right:Nn</code>	

<code>\tl_build_put_right:Ne</code>	

<code>\tl_build_gput_right:Nn</code>	

<code>\tl_build_gput_right:Ne</code>	

Adds $\langle tokens \rangle$ to the left or right side of the current contents of $\langle tl\ var \rangle$. The $\langle tl\ var \rangle$ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global $\langle tl\ var \rangle$ respectively. The `right` functions are about twice faster than the `left` functions.

<code>\tl_build_end:N</code>	<code>\tl_build_end:N</code> $\langle tl\ var \rangle$
<code>\tl_build_gend:N</code>	Gets the contents of $\langle tl\ var \rangle$ and stores that into the $\langle tl\ var \rangle$ using <code>\tl_set:Nn</code> or <code>\tl_gset:Nn</code> . The $\langle tl\ var \rangle$ must have been set up with <code>\tl_build_begin:N</code> or <code>\tl_build_gbegin:N</code> . The <code>end</code> and <code>gend</code> functions must be used for local and global $\langle tl\ var \rangle$ respectively. These functions completely remove the setup code that enabled $\langle tl\ var \rangle$ to be used for other <code>\tl_build_...</code> functions. After the action of <code>end/gend</code> , the $\langle tl\ var \rangle$ may be manipulated using standard <code>tl</code> functions.

<code>\tl_build_get_intermediate:NN</code>	<code>\tl_build_get_intermediate:NN</code> $\langle tl\ var_1 \rangle$ $\langle tl\ var_2 \rangle$

New: 2023-12-14

Stores the contents of the $\langle tl\ var_1 \rangle$ in the $\langle tl\ var_2 \rangle$. The $\langle tl\ var_1 \rangle$ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The $\langle tl\ var_2 \rangle$ is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

Chapter 17

The `l3str` module Strings

`TeX` associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a `TeX` sense.

A `TeX` string (and thus an `expl3` string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a `TeX` string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialized token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

Most expandable functions in this module come in three flavors:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

17.1 Creating and initializing string variables

<code>\str_new:N</code>	<code>\str_new:N <str var></code>
<code>\str_new:c</code>	Creates a new <code><str var></code> or raises an error if the name is already taken. The declaration is global. The <code><str var></code> is initially empty.
<code>\str_const:Nn</code>	<code>\str_const:Nn <str var> {<token list>}</code>
<code>\str_const:(NV Ne cn cV ce)</code>	Creates a new constant <code><str var></code> or raises an error if the name is already taken. The value of the <code><str var></code> is set globally to the <code><token list></code> , converted to a string.
<code>\str_clear:N</code>	<code>\str_clear:N <str var></code>
<code>\str_clear:c</code>	Clears the content of the <code><str var></code> .
<code>\str_gclear:N</code>	
<code>\str_gclear:c</code>	
<code>\str_clear_new:N</code>	<code>\str_clear_new:N <str var></code>
<code>\str_clear_new:c</code>	Ensures that the <code><str var></code> exists globally by applying <code>\str_new:N</code> if necessary, then applies <code>\str_(g)clear:N</code> to leave the <code><str var></code> empty.
<code>\str_gclear_new:N</code>	
<code>\str_gclear_new:c</code>	
<code>\str_set_eq:NN</code>	<code>\str_set_eq:NN <str var₁₂</code>
<code>\str_set_eq:(cN Nc cc)</code>	Sets the content of <code><str var_{1 equal to that of <code><str var_{2.}</code>}</code>
<code>\str_gset_eq:NN</code>	
<code>\str_gset_eq:(cN Nc cc)</code>	
<code>\str_concat:NNN</code>	<code>\str_concat:NNN <str var₁₂₃</code>
<code>\str_concat:ccc</code>	Concatenates the content of <code><str var_{2 and <code><str var_{3 together and saves the result in <code><str var_{1. The <code><str var_{2 is placed at the left side of the new string variable. The <code><str var_{2 and <code><str var_{3 must indeed be strings, as this function does not convert their contents to a string.}</code>}</code>}</code>}</code>}</code>}</code>
<code>\str_gconcat:NNN</code>	
<code>\str_gconcat:ccc</code>	
<code>\str_if_exist_p:N</code> *	<code>\str_if_exist_p:N <str var></code>
<code>\str_if_exist_p:c</code> *	<code>\str_if_exist:NTF <str var> {<true code>} {<false code>}</code>
<code>\str_if_exist:NTF</code> *	Tests whether the <code><str var></code> is currently defined. This does not check that the <code><str var></code> really is a string.
<code>\str_if_exist:cTF</code> *	

17.2 Adding data to string variables

<code>\str_set:Nn</code>	<code>\str_set:Nn <str var> {<token list>}</code>
<code>\str_set:(NV Ne cn cV ce)</code>	Converts the <code><token list></code> to a <code><string></code> , and stores the result in <code><str var></code> .
<code>\str_gset:Nn</code>	
<code>\str_gset:(NV Ne cn cV ce)</code>	

<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn <str var> {<token list>}</code>
<code>\str_put_left:(NV Ne cn cV ce)</code>	
<code>\str_gput_left:Nn</code>	
<code>\str_gput_left:(NV Ne cn cV ce)</code>	

Converts the `<token list>` to a `<string>`, and prepends the result to `<str var>`. The current contents of the `<str var>` are not automatically converted to a string.

<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn <str var> {<token list>}</code>
<code>\str_put_right:(NV Ne cn cV ce)</code>	
<code>\str_gput_right:Nn</code>	
<code>\str_gput_right:(NV Ne cn cV ce)</code>	

Converts the `<token list>` to a `<string>`, and appends the result to `<str var>`. The current contents of the `<str var>` are not automatically converted to a string.

17.3 String conditionals

<code>\str_if_empty_p:N *</code>	<code>\str_if_empty_p:N <str var></code>
<code>\str_if_empty_p:c *</code>	<code>\str_if_empty:NNTF <str var> {<true code>} {<false code>}</code>
<code>\str_if_empty:NNTF *</code>	
<code>\str_if_empty:cTF *</code>	
<code>\str_if_empty_p:n *</code>	
<code>\str_if_empty:nTF *</code>	

Updated: 2022-03-21

<code>\str_if_eq_p:NN *</code>	<code>\str_if_eq_p:NN <str var₁₂</code>
<code>\str_if_eq_p:(Nc cN cc) *</code>	<code>\str_if_eq:NNTF <str var₁₂</code>
<code>\str_if_eq:NNTF *</code>	
<code>\str_if_eq:(Nc cN cc)TF *</code>	

Compares the content of two `<str variables>` and is logically `true` if the two contain the same characters in the same order. See `\tl_if_eq:NNTF` to compare tokens (including their category codes) rather than characters.

<code>\str_if_eq_p:nn *</code>	<code>\str_if_eq_p:nn {<tl₁₂</code>
<code>\str_if_eq_p:(Vn on no nV VV vn nv ee) *</code>	<code>\str_if_eq:nnTF {<tl₁₂</code>
<code>\str_if_eq:nnTF *</code>	
<code>\str_if_eq:(Vn on no nV VV vn nv ee)TF *</code>	

Compares the two `<token lists>` on a character by character basis (namely after converting them to strings), and is `true` if the two `<strings>` contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically `true`. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

<code>\str_if_in:NnTF</code>	<code>\str_if_in:NnTF <str var> {<token list>} {<true code>} {<false code>}</code>
<code>\str_if_in:cnTF</code>	

Converts the `<token list>` to a `<string>` and tests if that `<string>` is found in the content of the `<str var>`.

```
\str_if_in:nnTF \str_if_in:nnTF {<tl1>} {<tl2>} {<true code>} {<false code>}
```

Converts both $\langle token\ lists \rangle$ to $\langle strings \rangle$ and tests whether $\langle string_2 \rangle$ is found inside $\langle string_1 \rangle$.

```
\str_case:nn      * \str_case:nnTF {<test string>}
\str_case:(Vn|on|en|nV|nv|ne) * {
\str_case:nnTF    *   {<string case1>} {<code case1>}
\str_case:(Vn|on|en|nV|nv|ne)TF *   {<string case2>} {<code case2>}
\str_case:Nn      *   ...
\str_case:NnTF    *   {<string casen>} {<code casen>}
\str_case:NnTF    *   }
Updated: 2022-03-21 {<true code>}
                    {<false code>}
```

Compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ case \rangle$ s until a match is found (all token lists are converted to strings). If the two are equal (as described for $\backslash\text{str\_if\_eq:nnTF}$) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\backslash\text{str\_case:nn}$, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each $\langle string\ case \rangle$ argument, so any variable in there will be compared as a string. If expansion is needed in the $\langle string\ case \rangle$ s, then $\backslash\text{str\_case\_e:nn(TF)}$ should be used instead.

```
\str_case_e:nn    * \str_case_e:nnTF {<test string>}
\str_case_e:en    * {
\str_case_e:nnTF  *   {<string case1>} {<code case1>}
\str_case_e:enTF  *   {<string case2>} {<code case2>}
\str_case_e:nnTF  *   ...
\str_case_e:enTF  *   {<string casen>} {<code casen>}
\str_case_e:nnTF  *   }
\str_case_e:nnTF  *   {<true code>}
\str_case_e:nnTF  *   {<false code>}
```

Compares the full expansion of the $\langle test\ string \rangle$ in turn with the full expansion of the $\langle string\ case \rangle$ s (all token lists are converted to strings). If the two full expansions are equal (as described for $\backslash\text{str\_if\_eq:eeTF}$) then the associated $\langle code \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function $\backslash\text{str\_case\_e:nn}$, which does nothing if there is no match, is also available. In $\backslash\text{str\_case\_e:nn(TF)}$, the $\langle test\ string \rangle$ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

<code>\str_compare_p:nNn</code>	★	<code>\str_compare_p:nNn {<tl₁>} <relation> {<tl₂>}</code>
<code>\str_compare_p:eNe</code>	★	<code>\str_compare:nNnTF {<tl₁>} <relation> {<tl₂>} {<true code>} {<false code>}</code>
<code>\str_compare:nNnTF</code>	★	Compares the two <i><token lists></i> on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The <i><relation></i> can be <i><</i> , <i>=</i> , or <i>></i> and the test is <i>true</i> under the following conditions:
<code>\str_compare:eNeTF</code>	★	

New: 2021-05-17

- for *<*, if the first string is earlier than the second in lexicographic order;
- for *=*, if the two strings have exactly the same characters;
- for *>*, if the first string is later than the second in lexicographic order.

Thus for example the following is logically *true*:

```
\str_compare_p:nNn { ab } < { abc }
```

T_EXhackers note: This is a wrapper around the T_EX primitive `\(pdf)strcmp`. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

17.4 Mapping over strings

All mappings are done at the current group level, i.e., any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

<code>\str_map_function:nN</code>	☆	<code>\str_map_function:nN {<token list>} <function></code>
<code>\str_map_function:NN</code>	☆	<code>\str_map_function:NN <str var> <function></code>
<code>\str_map_function:cN</code>	☆	Converts the <i><token list></i> to a <i><string></i> then applies <i><function></i> to every <i><character></i> in the <i><string></i> including spaces.

<code>\str_map_inline:nN</code>		<code>\str_map_inline:nN {<token list>} {<inline function>}</code>
<code>\str_map_inline:Nn</code>		<code>\str_map_inline:Nn <str var> {<inline function>}</code>
<code>\str_map_inline:cN</code>		Converts the <i><token list></i> to a <i><string></i> then applies the <i><inline function></i> to every <i><character></i> in the <i><str var></i> including spaces. The <i><inline function></i> should consist of code which receives the <i><character></i> as #1.

<code>\str_map_tokens:nN</code>	☆	<code>\str_map_tokens:nN {<token list>} {<code>}</code>
<code>\str_map_tokens:Nn</code>	☆	<code>\str_map_tokens:Nn <str var> {<code>}</code>
<code>\str_map_tokens:cN</code>	☆	Converts the <i><token list></i> to a <i><string></i> then applies <i><code></i> to every <i><character></i> in the <i><string></i> including spaces. The <i><code></i> receives each character as a trailing brace group. This is equivalent to <code>\str_map_function:nN</code> if the <i><code></i> consists of a single function.

New: 2021-05-05

<code>\str_map_variable:nNn</code>	<code>\str_map_variable:nNn {⟨token list⟩} ⟨variable⟩ {⟨code⟩}</code>
<code>\str_map_variable:NNn</code>	<code>\str_map_variable:NNn ⟨str var⟩ ⟨variable⟩ {⟨code⟩}</code>
<code>\str_map_variable:cNn</code>	Converts the <code>⟨token list⟩</code> to a <code>⟨string⟩</code> then stores each <code>⟨character⟩</code> in the <code>⟨string⟩</code> (including spaces) in turn in the (string or token list) <code>⟨variable⟩</code> and applies the <code>⟨code⟩</code> . The <code>⟨code⟩</code> will usually make use of the <code>⟨variable⟩</code> , but this is not enforced. The assignments to the <code>⟨variable⟩</code> are local. Its value after the loop is the last <code>⟨character⟩</code> in the <code>⟨string⟩</code> , or its original value if the <code>⟨string⟩</code> is empty. See also <code>\str_map-inline:Nn</code> .

<code>\str_map_break: ☆</code>	<code>\str_map_break:</code>

Used to terminate a `\str_map...` function before all characters in the `⟨string⟩` have been processed. This normally takes place within a conditional statement, for example

```

\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
  % Do something useful
}

```

See also `\str_map_break:n`. Use outside of a `\str_map...` scenario leads to low level $\text{\TeX}$ errors.

$\text{\TeX}$ hackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

<code>\str_map_break:n ☆</code>	<code>\str_map_break:n {⟨code⟩}</code>

Used to terminate a `\str_map...` function before all characters in the `⟨string⟩` have been processed, inserting the `⟨code⟩` after the mapping has ended. This normally takes place within a conditional statement, for example

```

\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo }
  { \str_map_break:n { <code> } }
  % Do something useful
}

```

Use outside of a `\str_map...` scenario leads to low level $\text{\TeX}$ errors.

$\text{\TeX}$ hackers note: When the mapping is broken, additional tokens may be inserted before the `⟨code⟩` is inserted into the input stream. This depends on the design of the mapping function.

17.5 Working with the content of strings

<code>\str_use:N</code>	★	<code>\str_use:N <str var></code>
<code>\str_use:c</code>	★	

Recovers the content of a `<str var>` and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a `<str>` directly without an accessor function.

<code>\str_count:N</code>	★	<code>\str_count:n {<token list>}</code>
<code>\str_count:c</code>	★	
<code>\str_count:n</code>	★	
<code>\str_count:e</code>	★	
<code>\str_count_ignore_spaces:n</code>	★	
<code>\str_count_ignore_spaces:e</code>	★	

Leaves in the input stream the number of characters in the string representation of `<token list>`, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

<code>\str_count_spaces:N</code>	★	<code>\str_count_spaces:n {<token list>}</code>
<code>\str_count_spaces:c</code>	★	
<code>\str_count_spaces:n</code>	★	

Leaves in the input stream the number of space characters in the string representation of `<token list>`, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {<token list>}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

Converts the `<token list>` into a `<string>`. The first character in the `<string>` is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the `<string>` is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {<token list>}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

Converts the `<token list>` to a `<string>`, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the `<token list>` is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

<code>\str_item:Nn</code>	<code>*</code>	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:cn</code>	<code>*</code>	
<code>\str_item:nn</code>	<code>*</code>	
<code>\str_item_ignore_spaces:nn</code>	<code>*</code>	

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer\ expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the $\langle integer\ expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, etc.

<code>\str_range:Nnn</code>	<code>*</code>	<code>\str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}</code>
<code>\str_range:cn</code>	<code>*</code>	
<code>\str_range:nnn</code>	<code>*</code>	
<code>\str_range_ignore_spaces:nnn</code>	<code>*</code>	

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here $\langle start\ index \rangle$ and $\langle end\ index \rangle$ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```
\iow_term:e { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { 0 } { -1 } }
```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The $\langle start\ index \rangle$ must always be smaller than or equal to the $\langle end\ index \rangle$: if this is not the case then no output is generated. Thus

```
\iow_term:e { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:e { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { 5 } }
```

```

\iow_term:e { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:e { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }

```

will print four instances of bcde, four instances of bc e and eight instances of bcde.

17.6 Modifying string variables

<code>\str_replace_once:Nnn</code>	<code>\str_replace_once:Nnn <str var> {<old>} {<new>}</code>
<code>\str_replace_once:cnn</code>	Converts the <code><old></code> and <code><new></code> token lists to strings, then replaces the first (leftmost)
<code>\str_greplace_once:Nnn</code>	occurrence of <code><old string></code> in the <code><str var></code> with <code><new string></code> .
<code>\str_greplace_once:cnn</code>	

<code>\str_replace_all:Nnn</code>	<code>\str_replace_all:Nnn <str var> {<old>} {<new>}</code>
<code>\str_replace_all:cnn</code>	Converts the <code><old></code> and <code><new></code> token lists to strings, then replaces all occurrences of <code><old</code>
<code>\str_greplace_all:Nnn</code>	<code>string></code> in the <code><str var></code> with <code><new string></code> . As this function operates from left to
<code>\str_greplace_all:cnn</code>	right, the pattern <code><old string></code> may remain after the replacement (see <code>\str_remove_all:Nn</code> for an example).

<code>\str_remove_once:Nn</code>	<code>\str_remove_once:Nn <str var> {<token list>}</code>
<code>\str_remove_once:cn</code>	Converts the <code><token list></code> to a <code><string></code> then removes the first (leftmost) occurrence
<code>\str_gremove_once:Nn</code>	of <code><string></code> from the <code><str var></code> .
<code>\str_gremove_once:cn</code>	

<code>\str_remove_all:Nn</code>	<code>\str_remove_all:Nn <str var> {<token list>}</code>
<code>\str_remove_all:cn</code>	Converts the <code><token list></code> to a <code><string></code> then removes all occurrences of <code><string></code> from
<code>\str_gremove_all:Nn</code>	the <code><str var></code> . As this function operates from left to right, the pattern <code><string></code> may
<code>\str_gremove_all:cn</code>	remain after the removal, for instance,

```

\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}

```

results in `\l_tmpa_str` containing `abcd`.

17.7 String manipulation

```
\str_lowercase:n * \str_lowercase:n {(tokens)}  
\str_lowercase:f * \str_uppercase:n {(tokens)}  
\str_uppercase:n *  
\str_uppercase:f *
```

Converts the input `{tokens}` to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2  
{  
  \cs_set_protected:cpn  
  {  
    user  
    \str_uppercase:f { \tl_head:n {#1} }  
    \str_lowercase:f { \tl_tail:n {#1} }  
  }  
  { #2 }  
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_casefold:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase_(all|first):n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

<code>\str_casefold:n</code> *	<code>\str_casefold:n {<tokens>}</code>
<code>\str_casefold:V</code> *	Converts the input <code><tokens></code> to their string representation, as described for <code>\tl_to_str:n</code> , and then folds the case of the resulting <code><string></code> to remove case information. The result of this process is left in the input stream.
New: 2022-10-16	

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_casefold:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_casefold:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* `SS` and `ß` both fold to `ss`). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.*, `I` always folds to `i` and not to `ı`).

<code>\str_md5hash:n</code> *	<code>\str_md5hash:n {<tokens>}</code>
<code>\str_md5hash:(V v o e)</code> *	Expands to the MD5 sum generated from the <code><tokens></code> , which is converted to a <code><string></code> as described for <code>\tl_to_str:n</code> .
New: 2023-05-19	

17.8 Viewing strings

<code>\str_show:N</code>	<code>\str_show:N <str var></code>
<code>\str_show:c</code>	Displays the content of the <code><str var></code> on the terminal.
<code>\str_show:n</code>	
Updated: 2021-04-29	
<code>\str_log:N</code>	<code>\str_log:N <str var></code>
<code>\str_log:c</code>	Writes the content of the <code><str var></code> in the log file.
<code>\str_log:n</code>	
Updated: 2021-04-29	

17.9 Constant strings

<code>\c_ampersand_str</code>	Constant strings, containing a single character token, with category code 12.
<code>\c_atsign_str</code>	
<code>\c_backslash_str</code>	
<code>\c_left_brace_str</code>	
<code>\c_right_brace_str</code>	
<code>\c_circumflex_str</code>	
<code>\c_colon_str</code>	
<code>\c_dollar_str</code>	
<code>\c_hash_str</code>	
<code>\c_percent_str</code>	
<code>\c_tilde_str</code>	
<code>\c_underscore_str</code>	
<code>\c_zero_str</code>	

Updated: 2020-12-22

<code>\c_empty_str</code>	Constant that is always empty.

New: 2023-12-07

17.10 Scratch strings

<code>\l_tmpa_str</code>	Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_str</code>	

<code>\g_tmpa_str</code>	Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_str</code>	

Chapter 18

The l3str-convert module

String encoding conversions

18.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, etc. See Table 1 for a list of supported encodings.⁶
- Bytes are translated to T_EX tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.⁶

⁶Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<i>⟨Encoding⟩</i>	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<code>⟨empty⟩</code>	native (Unicode) string
<code>default</code>	like <code>utf8</code> with 8-bit engines, and like native with unicode-engines

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<i>⟨Escaping⟩</i>	description
<code>bytes</code> , or <code>empty</code>	arbitrary bytes
<code>hex</code> , <code>hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescapename</code>
<code>string</code>	see <code>\pdfescapestring</code>
<code>url</code>	encoding used in URLs

18.2 Conversion functions

<code>\str_set_convert:Nnnn</code> <code>\str_gset_convert:Nnnn</code>	<code>\str_set_convert:Nnnn <str var> {<string>} {<name₁>} {<name₂>}</code>

This function converts the *<string>* from the encoding given by *<name₁>* to the encoding given by *<name₂>*, and stores the result in the *<str var>*. Each *<name>* can have the form *<encoding>* or *<encoding>/<escaping>*, where the possible values of *<encoding>* and *<escaping>* are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty *<name>* indicates the use of “native” strings, 8-bit for pdfTeX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “FEFF”, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the *<string>* is not valid according to the *<escaping 1>* and *<encoding 1>*, or if it cannot be reencoded in the *<encoding 2>* and *<escaping 2>* (for instance, if a character does not exist in the *<encoding 2>*). Erroneous input is replaced by the Unicode replacement character “FFFD”, and characters which cannot be reencoded are replaced by either the replacement character “FFFD” if it exists in the *<encoding 2>*, or an encoding-specific replacement character, or the question mark character.

<code>\str_set_convert:NnnnTF</code> <code>\str_gset_convert:NnnnTF</code>	<code>\str_set_convert:NnnnTF <str var> {<string>} {<name₁>} {<name₂>} {<true code>}</code> <code>{<false code>}</code>

As `\str_set_convert:Nnnn`, converts the *<string>* from the encoding given by *<name₁>* to the encoding given by *<name₂>*, and assigns the result to *<str var>*. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the *<string>* is not valid according to the *<name₁>* encoding, or cannot be expressed in the *<name₂>* encoding. Instead, the *<false code>* is performed.

18.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

<code>\str_convert_pdfname:n</code> ★	<code>\str_convert_pdfname:n {<string>}</code>

As `\str_set_convert:Nnnn`, converts the *<string>* on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

18.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In XeTeX/LuaTeX, would it be better to use the `^^^...` approach to build a string from a given list of character codes? Namely, within a group, assign 0-9a-f and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in [“D800,”DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! ' ( ) * - . / 0 1 2 3 4 5 6 7 8 9 _` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko’s `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, etc.?

Chapter 19

The l3str-format package

Formatting strings of characters

19.1 Format specifications

L^AT_EX3 has the notion of a string $\langle format \rangle$. The syntax follows that of Python's `format` built-in function. A $\langle format specification \rangle$ is a string of the form

$$\langle format specification \rangle = [[\langle fill \rangle][\langle alignment \rangle]][\langle sign \rangle][\langle width \rangle][.\langle precision \rangle][\langle style \rangle]$$

where each [...] denotes an independent optional part.

- $\langle fill \rangle$ can be any character: it is assumed to be present whenever the second character of the $\langle format specification \rangle$ is a valid $\langle alignment \rangle$ character.
- $\langle alignment \rangle$ can be < (left alignment), > (right alignment), ^ (centering), or = (for numeric types only).
- $\langle sign \rangle$ is allowed for numeric types; it can be + (show a sign for positive and negative numbers), - (only put a sign for negative numbers), or a space (show a space or a -).
- $\langle width \rangle$ is the minimum number of characters of the result: if the result is naturally shorter than this $\langle width \rangle$, then it is padded with copies of the character $\langle fill \rangle$, with a position depending on the choice of $\langle alignment \rangle$. If the result is naturally longer, it is not truncated.
- $\langle precision \rangle$, whose presence is indicated by a period, can have different meanings depending on the type.
- $\langle style \rangle$ is one character, which controls how the given data should be formatted. The list of allowed $\langle styles \rangle$ depends on the type.

The choice of $\langle alignment \rangle =$ is only valid for numeric types: in this case the padding is inserted between the sign and the rest of the number.

Details of the individual formatting functions are given in the relevant modules, e.g. `l3fp` for `\fp_format:nn`.

Chapter 20

The l3quark module

Quarks and scan marks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

20.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (i.e., `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

20.2 Defining quarks

<u><code>\quark_new:N</code></u>	<code>\quark_new:N <quark></code> Creates a new <code><quark></code> which expands only to <code><quark></code> . The <code><quark></code> is defined globally, and an error message is raised if the name was already taken.
<u><code>\q_stop</code></u>	Used as a marker for delimited arguments, such as <code>\cs_set:Npn \tmp:w #1#2 \q_stop {#1}</code>
<u><code>\q_mark</code></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><code>\q_nil</code></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><code>\q_no_value</code></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

20.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

<u><code>\quark_if_nil_p:N</code></u>	<code>\quark_if_nil_p:N <token></code>
<u><code>\quark_if_nil:NTF</code></u>	<code>\quark_if_nil:NTF <token> {<true code>} {<false code>}</code> Tests if the <code><token></code> is equal to <code>\q_nil</code> .
<u><code>\quark_if_nil_p:n</code></u>	<code>\quark_if_nil_p:n {<token list>}</code>
<u><code>\quark_if_nil_p:(o V)</code></u>	<code>\quark_if_nil:nTF {<token list>} {<true code>} {<false code>}</code>
<u><code>\quark_if_nil:nTF</code></u>	<code>\quark_if_nil:nTF <token list> {<true code>} {<false code>}</code>
<u><code>\quark_if_nil:(o V)TF</code></u>	<code>\quark_if_nil:(o V)TF <token list> {<true code>} {<false code>}</code> Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or containing <code>\q_nil</code> plus one or more other tokens).
<u><code>\quark_if_no_value_p:N</code></u>	<code>\quark_if_no_value_p:N <token></code>
<u><code>\quark_if_no_value_p:c</code></u>	<code>\quark_if_no_value_p:c <token> {<true code>} {<false code>}</code>
<u><code>\quark_if_no_value:NTF</code></u>	<code>\quark_if_no_value:NTF <token> {<true code>} {<false code>}</code>
<u><code>\quark_if_no_value:cTF</code></u>	<code>\quark_if_no_value:cTF <token> {<true code>} {<false code>}</code> Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><code>\quark_if_no_value_p:n</code></u>	<code>\quark_if_no_value_p:n {<token list>}</code>
<u><code>\quark_if_no_value:nTF</code></u>	<code>\quark_if_no_value:nTF {<token list>} {<true code>} {<false code>}</code> Tests if the <code><token list></code> contains only <code>\q_no_value</code> (distinct from <code><token list></code> being empty or containing <code>\q_no_value</code> plus one or more other tokens).

20.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 20.4.1.

`\q_recursion_tail` This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop` This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N` `\quark_if_recursion_tail_stop:N <token>`

Tests if `<token>` contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n` `\quark_if_recursion_tail_stop:n {(token list)}`

`\quark_if_recursion_tail_stop:o` `\quark_if_recursion_tail_stop:o`

Tests if the `<token list>` contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn` `\quark_if_recursion_tail_stop_do:Nn <token> {(insertion)}`

Tests if `<token>` contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The `<insertion>` code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn` `\quark_if_recursion_tail_stop_do:nn {(token list)} {(insertion)}`

`\quark_if_recursion_tail_stop_do:on` `\quark_if_recursion_tail_stop_do:on`

Tests if the `<token list>` contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The `<insertion>` code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_break:NN` `\quark_if_recursion_tail_break:NN {(token list)} \<type>_map_break:`

`\quark_if_recursion_tail_break:nN` `\quark_if_recursion_tail_break:nN`

Tests if `<token list>` contains only `\q_recursion_tail`, and if so terminates the recursion using `\<type>_map_break:.` The recursion end should be marked by `\prg_break_point:Nn \<type>_map_break:.`

20.4.1 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “`[-a-b-] [-c-d-]`”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
{
  \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
  \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
  \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
{
  \quark_if_recursion_tail_stop:n {#1}
  \quark_if_recursion_tail_stop:n {#2}
  \__my_map_dbl_fn:nn {#1} {#2}
}
```

Finally, recurse:

```
  \__my_map_dbl:nn
}
```

Note that contrarily to $\text{\LaTeX}3$ built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `\__my_map_dbl_fn:nn`.

20.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by $\text{\TeX}$ in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `l3regex`).

`\scan_new:N` `\scan_new:N` $\langle scan\ mark \rangle$

Creates a new $\langle scan\ mark \rangle$ which is set equal to `\scan_stop:`. The $\langle scan\ mark \rangle$ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s_stop` Used at the end of a set of instructions, as a marker that can be jumped to using `\use_none_delimit_by_s_stop:w`.

`\use_none_delimit_by_s_stop:w` $\star$ `\use_none_delimit_by_s_stop:w` $\langle tokens \rangle$ `\s_stop`

Removes the $\langle tokens \rangle$ and `\s_stop` from the input stream. This leads to a low-level $\TeX$ error if `\s_stop` is absent.

Chapter 21

The l3seq module

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *⟨balanced text⟩*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

21.1 Creating and initializing sequences

<code>\seq_new:N</code>	<code>\seq_new:N <seq var></code>
<code>\seq_new:c</code>	Creates a new <i>⟨seq var⟩</i> or raises an error if the name is already taken. The declaration is global. The <i>⟨seq var⟩</i> initially contains no items.

<code>\seq_clear:N</code>	<code>\seq_clear:N <seq var></code>
<code>\seq_clear:c</code>	Clears all items from the <i>⟨seq var⟩</i> .
<code>\seq_gclear:N</code>	
<code>\seq_gclear:c</code>	

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N <seq var></code>
<code>\seq_clear_new:c</code>	Ensures that the <i>⟨seq var⟩</i> exists globally by applying <code>\seq_new:N</code> if necessary, then applies <code>\seq_(g)clear:N</code> to leave the <i>⟨seq var⟩</i> empty.
<code>\seq_gclear_new:N</code>	
<code>\seq_gclear_new:c</code>	

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN <seq var₁> <seq var₂></code>
<code>\seq_set_eq:(cN Nc cc)</code>	Sets the content of <i>⟨seq var₁⟩</i> equal to that of <i>⟨seq var₂⟩</i> .
<code>\seq_gset_eq:NN</code>	
<code>\seq_gset_eq:(cN Nc cc)</code>	

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN <seq var> <clist var></code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

Converts the data in the `<clist var>` into a `<seq var>`: the original `<clist var>` is unchanged.

<code>\seq_const_from_clist:Nn</code>	<code>\seq_const_from_clist:Nn <seq var> {<comma-list>}</code>
<code>\seq_const_from_clist:cn</code>	

Creates a new constant `<seq var>` or raises an error if the name is already taken. The `<seq var>` is set globally to contain the items in the `<comma list>`.

<code>\seq_set_split:Nnn</code>	<code>\seq_set_split:Nnn <seq var> {<delimiter>} {<token list>}</code>
<code>\seq_set_split:(NVn NnV NVV Nne Nee)</code>	
<code>\seq_gset_split:Nnn</code>	
<code>\seq_gset_split:(NVn NnV NVV Nne Nee)</code>	

Splits the `<token list>` into `<items>` separated by `<delimiter>`, and assigns the result to the `<seq var>`. Spaces on both sides of each `<item>` are ignored, then one set of outer braces is removed (if any); this space trimming behavior is identical to that of `\l3clist` functions. Empty `<items>` are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <seq var> {}`. The `<delimiter>` may not contain `{, }` or `#` (assuming T_EX's normal category code régime). If the `<delimiter>` is empty, the `<token list>` is split into `<items>` as described for `\tl_map_function:nN`. See also `\seq_set_split_keep_spaces:Nnn`, which omits space stripping.

<code>\seq_set_split_keep_spaces:Nnn</code>	<code>\seq_set_split_keep_spaces:Nnn <seq var> {<delimiter>} {<token list>}</code>
<code>\seq_set_split_keep_spaces:NnV</code>	
<code>\seq_gset_split_keep_spaces:Nnn</code>	
<code>\seq_gset_split_keep_spaces:NnV</code>	

New: 2021-03-24

Splits the `<token list>` into `<items>` separated by `<delimiter>`, and assigns the result to the `<seq var>`. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty `<items>` are preserved by `\seq_set_split_keep_spaces:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <seq var> {}`. The `<delimiter>` may not contain `{, }` or `#` (assuming T_EX's normal category code régime). If the `<delimiter>` is empty, the `<token list>` is split into `<items>` as described for `\tl_map_function:nN`; note in this case spaces will *not* be preserved. See also `\seq_set_split:Nnn`, which removes spaces around the delimiters.

<code>\seq_set_filter:Nnn</code>	<code>\seq_set_filter:Nnn <seq var₁> <seq var₂> {<inline boolexpr>}</code>
<code>\seq_gset_filter:NNn</code>	Evaluates the <code><inline boolexpr></code> for every <code><item></code> stored within the <code><seq var₂></code> . The <code><inline boolexpr></code> receives the <code><item></code> as #1. The sequence of all <code><items></code> for which the <code><inline boolexpr></code> evaluated to <code>true</code> is assigned to <code><seq var₁></code> .

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

<code>\seq_set_regex_extract_once:Nnn</code>	<code>\seq_set_regex_extract_once:Nnn <seq var> {<regex>} {<token list>}</code>
<code>\seq_set_regex_extract_once:cnn</code>	<code>\seq_set_regex_extract_once:NNn <seq var> <regex var> {<token list>}</code>
<code>\seq_set_regex_extract_once:NNn</code>	
<code>\seq_set_regex_extract_once:cNn</code>	
<code>\seq_gset_regex_extract_once:Nnn</code>	
<code>\seq_gset_regex_extract_once:cnn</code>	
<code>\seq_gset_regex_extract_once:NNn</code>	
<code>\seq_gset_regex_extract_once:cNn</code>	

New: 2024-12-08

Finds the first match of the `<regex>` in the `<token list>`. If it exists, the match is stored as the first item of the `<seq var>`, and further items are the contents of capturing groups, in the order of their opening parenthesis. If there is no match, the `<seq var>` is cleared. Theses are alternative names for `\regex_extract_once:nnN` and friends, with arguments re-ordered for `<seq var>` setting; see `l3regex` chapter for more details of the `<regex>` format.

<code>\seq_set_regex_extract_all:Nnn</code>	<code>\seq_set_regex_extract_all:Nnn <seq var> {<regex>} {<token list>}</code>
<code>\seq_set_regex_extract_all:cnn</code>	<code>\seq_set_regex_extract_all:NNn <seq var> <regex var> {<token list>}</code>
<code>\seq_set_regex_extract_all:NNn</code>	
<code>\seq_set_regex_extract_all:cNn</code>	
<code>\seq_gset_regex_extract_all:Nnn</code>	
<code>\seq_gset_regex_extract_all:cnn</code>	
<code>\seq_gset_regex_extract_all:NNn</code>	
<code>\seq_gset_regex_extract_all:cNn</code>	

New: 2024-12-08

Finds all matches of the `<regex>` in the `<token list>`, and stores all the submatch information in a single sequence (concatenating the results of multiple `\seq_set_regex_extract_all:Nnn` calls). If there is no match, the `<seq var>` is cleared. Theses are alternative names for `\regex_extract_all:nnN` and friends, with arguments re-ordered for `<seq var>` setting; see `l3regex` chapter for more details of the `<regex>` format.

```

\seq_set_regex_split:Nnn
\seq_set_regex_split:cnn
\seq_set_regex_split:NNn
\seq_set_regex_split:cNn
\seq_gset_regex_split:Nnn
\seq_gset_regex_split:cnn
\seq_gset_regex_split:NNn
\seq_gset_regex_split:cNn

```

New: 2024-12-08

```

\seq_set_regex_split:Nnn <seq var> {(regex)} {(token list)}
\seq_set_regex_split:NNn <seq var> <regex var> {(token list)}

```

Splits the $\langle token\ list \rangle$ into a sequence of parts, delimited by matches of the $\langle regular\ expression \rangle$. If the $\langle regular\ expression \rangle$ has capturing groups, then the token lists that they match are stored as items of the sequence as well. If no match is found the resulting $\langle seq\ var \rangle$ has the $\langle token\ list \rangle$ as its sole item. If the $\langle regular\ expression \rangle$ matches the empty token list, then the $\langle token\ list \rangle$ is split into single tokens. For example, after

```
\seq_set_regex_split:Nnn \l_path_seq { / } { the/path/for/this/file.tex }
```

the sequence $\l_path_seq$ contains the items $\{the\}$, $\{path\}$, $\{for\}$, $\{this\}$, and $\{file.tex\}$. Theses are alternative names for $\backslash regex\_split:nnN$ and friends, with arguments re-ordered for $\langle seq\ var \rangle$ setting; see `l3regex` chapter for more details of the $\langle regex \rangle$ format.

```

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

```

```
\seq_concat:NNN <seq var1> <seq var2> <seq var3>
```

Concatenates the content of $\langle seq\ var_2 \rangle$ and $\langle seq\ var_3 \rangle$ together and saves the result in $\langle seq\ var_1 \rangle$. The items in $\langle seq\ var_2 \rangle$ are placed at the left side of the new sequence.

```

\seq_if_exist_p:N * \seq_if_exist_p:N <seq var>
\seq_if_exist_p:c * \seq_if_exist:NTF <seq var> {(true code)} {(false code)}
\seq_if_exist:NTF * \seq_if_exist:cTF *

```

Tests whether the $\langle seq\ var \rangle$ is currently defined. This does not check that the $\langle seq\ var \rangle$ really is a sequence variable.

21.2 Appending data to sequences

```

\seq_put_left:Nn
\seq_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

```

Appends the $\langle item \rangle$ to the left of the $\langle seq\ var \rangle$.

```

\seq_put_right:Nn
\seq_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

```

Appends the $\langle item \rangle$ to the right of the $\langle seq\ var \rangle$.

21.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, i.e., setting the $\langle tl\ var \rangle$ used with $\backslash tl\_set:Nn$ and *never* $\backslash tl\_gset:Nn$.

<code>\seq_get_left:NN</code>	<code>\seq_get_left:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_get_left:cN</code>	Stores the left-most item from a $\langle seq\ var \rangle$ in the $\langle t1\ var \rangle$ without removing it from the $\langle seq\ var \rangle$. The $\langle t1\ var \rangle$ is assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_get_right:NN</code>	<code>\seq_get_right:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_get_right:cN</code>	Stores the right-most item from a $\langle seq\ var \rangle$ in the $\langle t1\ var \rangle$ without removing it from the $\langle seq\ var \rangle$. The $\langle t1\ var \rangle$ is assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_pop_left:NN</code>	<code>\seq_pop_left:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_pop_left:cN</code>	Pops the left-most item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$, i.e., removes the item from the sequence and stores it in the $\langle t1\ var \rangle$. Both of the variables are assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_gpop_left:NN</code>	<code>\seq_gpop_left:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_gpop_left:cN</code>	Pops the left-most item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$, i.e., removes the item from the sequence and stores it in the $\langle t1\ var \rangle$. The $\langle seq\ var \rangle$ is modified globally, while the assignment of the $\langle t1\ var \rangle$ is local. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_pop_right:NN</code>	<code>\seq_pop_right:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_pop_right:cN</code>	Pops the right-most item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$, i.e., removes the item from the sequence and stores it in the $\langle t1\ var \rangle$. Both of the variables are assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_gpop_right:NN</code>	<code>\seq_gpop_right:NN</code> $\langle seq\ var \rangle$ $\langle t1\ var \rangle$
<code>\seq_gpop_right:cN</code>	Pops the right-most item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$, i.e., removes the item from the sequence and stores it in the $\langle t1\ var \rangle$. The $\langle seq\ var \rangle$ is modified globally, while the assignment of the $\langle t1\ var \rangle$ is local. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker <code>\q_no_value</code> .

<code>\seq_item:Nn</code>	$\star$ <code>\seq_item:Nn</code> $\langle seq\ var \rangle$ $\{ \langle integer\ expression \rangle \}$
<code>\seq_item:(NV Ne cn cV ce)</code>	$\star$ Indexing items in the $\langle seq\ var \rangle$ from 1 at the top (left), this function evaluates the $\langle integer\ expression \rangle$ and leaves the appropriate item from the sequence in the input stream. If the $\langle integer\ expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. If the $\langle integer\ expression \rangle$ is larger than the number of items in the $\langle seq\ var \rangle$ (as calculated by <code>\seq_count:N</code>) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an e-type or x-type argument expansion.

<u>\seq_rand_item:N</u> *	<u>\seq_rand_item:N</u> $\langle seq\ var \rangle$
<u>\seq_rand_item:c</u> *	Selects a pseudo-random item of the $\langle seq\ var \rangle$. If the $\langle seq\ var \rangle$ is empty the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an **e**-type or **x**-type argument expansion.

21.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

<u>\seq_get_left:NNTF</u>	<u>\seq_get_left:NNTF</u> $\langle seq\ var \rangle$ $\langle tl\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<u>\seq_get_left:cNTF</u>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle tl\ var \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, stores the left-most item from the $\langle seq\ var \rangle$ in the $\langle tl\ var \rangle$ without removing it from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle tl\ var \rangle$ is assigned locally.

<u>\seq_get_right:NNTF</u>	<u>\seq_get_right:NNTF</u> $\langle seq\ var \rangle$ $\langle tl\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<u>\seq_get_right:cNTF</u>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle tl\ var \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, stores the right-most item from the $\langle seq\ var \rangle$ in the $\langle tl\ var \rangle$ without removing it from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle tl\ var \rangle$ is assigned locally.

<u>\seq_pop_left:NNTF</u>	<u>\seq_pop_left:NNTF</u> $\langle seq\ var \rangle$ $\langle tl\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<u>\seq_pop_left:cNTF</u>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle tl\ var \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, pops the left-most item from the $\langle seq\ var \rangle$ in the $\langle tl\ var \rangle$, i.e., removes the item from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. Both the $\langle seq\ var \rangle$ and the $\langle tl\ var \rangle$ are assigned locally.

<u>\seq_gpop_left:NNTF</u>	<u>\seq_gpop_left:NNTF</u> $\langle seq\ var \rangle$ $\langle tl\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<u>\seq_gpop_left:cNTF</u>	If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle tl\ var \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, pops the left-most item from the $\langle seq\ var \rangle$ in the $\langle tl\ var \rangle$, i.e., removes the item from the $\langle seq\ var \rangle$, then leaves the $\langle true\ code \rangle$ in the input stream. The $\langle seq\ var \rangle$ is modified globally, while the $\langle tl\ var \rangle$ is assigned locally.

<code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNNTF</code>	<code>\seq_pop_right:NNTF <seq var> <tl var> {<true code>} {<false code>}</code> If the <code><seq var></code> is empty, leaves the <code><false code></code> in the input stream. The value of the <code><tl var></code> is not defined in this case and should not be relied upon. If the <code><seq var></code> is non-empty, pops the right-most item from the <code><seq var></code> in the <code><tl var></code> , i.e., removes the item from the <code><seq var></code> , then leaves the <code><true code></code> in the input stream. Both the <code><seq var></code> and the <code><tl var></code> are assigned locally.

<code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNNTF</code>	<code>\seq_gpop_right:NNTF <seq var> <tl var> {<true code>} {<false code>}</code> If the <code><seq var></code> is empty, leaves the <code><false code></code> in the input stream. The value of the <code><tl var></code> is not defined in this case and should not be relied upon. If the <code><seq var></code> is non-empty, pops the right-most item from the <code><seq var></code> in the <code><tl var></code> , i.e., removes the item from the <code><seq var></code> , then leaves the <code><true code></code> in the input stream. The <code><seq var></code> is modified globally, while the <code><tl var></code> is assigned locally.

21.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

<code>\seq_remove_duplicates:N</code> <code>\seq_remove_duplicates:c</code> <code>\seq_gremove_duplicates:N</code> <code>\seq_gremove_duplicates:c</code>	<code>\seq_remove_duplicates:N <seq var></code> Removes duplicate items from the <code><seq var></code> , leaving the left most copy of each item in the <code><seq var></code> . The <code><item></code> comparison takes place on a token basis, as for <code>\tl_if_eq:nnTF</code> .

TeXhackers note: This function iterates through every item in the `<seq var>` and does a comparison with the `<items>` already checked. It is therefore relatively slow with large sequences.

<code>\seq_remove_all:Nn</code> <code>\seq_remove_all:(NV Ne cn cV ce)</code> <code>\seq_gremove_all:Nn</code> <code>\seq_gremove_all:(NV Ne cn cV ce)</code>	<code>\seq_remove_all:Nn <seq var> {<item>}</code>

Removes every occurrence of `<item>` from the `<seq var>`. The `<item>` comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

<code>\seq_set_item:Nnn</code> <code>\seq_set_item:cnn</code> <code>\seq_set_item:NnnTF</code> <code>\seq_set_item:cnnTF</code> <code>\seq_gset_item:Nnn</code> <code>\seq_gset_item:cnn</code> <code>\seq_gset_item:NnnTF</code> <code>\seq_gset_item:cnnTF</code>	<code>\seq_set_item:Nnn <seq var> {<int expr>} {<item>}</code> <code>\seq_set_item:NnnTF <seq var> {<int expr>} {<item>} {<true code>} {<false code>}</code> Removes the item of <code><seq var></code> at the position given by evaluating the <code><int expr></code> and replaces it by <code><item></code> . Items are indexed from 1 on the left/top of the <code><seq var></code> , or from <code>-1</code> on the right/bottom. If the <code><int expr></code> is zero or is larger (in absolute value) than the number of items in the sequence, the <code><seq var></code> is not modified. In these cases, <code>\seq_set_item:Nnn</code> raises an error while <code>\seq_set_item:NnnTF</code> runs the <code><false code></code> . In cases where the assignment was successful, <code><true code></code> is run afterwards.

New: 2021-04-29

<code>\seq_reverse:N</code>	<code>\seq_reverse:N <seq var></code>
<code>\seq_reverse:c</code>	
<code>\seq_greverse:N</code>	Reverses the order of the items stored in the <code><seq var></code> .
<code>\seq_greverse:c</code>	
<code>\seq_sort:Nn</code>	<code>\seq_sort:Nn <seq var> {<comparison code>}</code>
<code>\seq_sort:cn</code>	
<code>\seq_gsort:Nn</code>	Sorts the items in the <code><seq var></code> according to the <code><comparison code></code> , and assigns the result to <code><seq var></code> . The details of sorting comparison are described in Section 6.1.
<code>\seq_gsort:cn</code>	
<code>\seq_shuffle:N</code>	<code>\seq_shuffle:N <seq var></code>
<code>\seq_shuffle:c</code>	
<code>\seq_gshuffle:N</code>	Sets the <code><seq var></code> to the result of placing the items of the <code><seq var></code> in a random order.
<code>\seq_gshuffle:c</code>	Each item is (roughly) as likely to end up in any given position.

T_EXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed`: only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

21.6 Sequence conditionals

<code>\seq_if_empty_p:N</code>	<code>\seq_if_empty_p:N <seq var></code>
<code>\seq_if_empty_p:c</code>	<code>\seq_if_empty:NNTF <seq var> {<true code>} {<false code>}</code>
<code>\seq_if_empty:NNTF</code>	<code>\seq_if_empty:NNTF <seq var> {<true code>} {<false code>}</code>
<code>\seq_if_empty:cTF</code>	Tests if the <code><seq var></code> is empty (containing no items).

<code>\seq_if_in:NnTF</code>	<code>\seq_if_in:NnTF <seq var> {<item>} {<true code>} {<false code>}</code>
<code>\seq_if_in:(NV Nv Ne No cn cV cv ce co)TF</code>	

Tests if the `<item>` is present in the `<seq var>`.

21.7 Mapping over sequences

All mappings are done at the current group level, i.e., any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

<code>\seq_map_function:NN</code>	<code>\seq_map_function:NN <seq var> <function></code>
<code>\seq_map_function:cN</code>	
	Applies <code><function></code> to every <code><item></code> stored in the <code><seq var></code> . The <code><function></code> will receive one argument for each iteration. The <code><items></code> are returned from left to right. To pass further arguments to the <code><function></code> , see <code>\seq_map_tokens:Nn</code> . The function <code>\seq_map_inline:Nn</code> is faster than <code>\seq_map_function:NN</code> for sequences with more than about 10 items.

<code>\seq_map_inline:Nn</code>	<code>\seq_map_inline:Nn <seq var> {<inline function>}</code>
<code>\seq_map_inline:cn</code>	
	Applies <code><inline function></code> to every <code><item></code> stored within the <code><seq var></code> . The <code><inline function></code> should consist of code which will receive the <code><item></code> as #1. The <code><items></code> are returned from left to right.

`\seq_map_tokens:Nn` ☆ `\seq_map_tokens:Nn <seq var> {<code>}`

`\seq_map_tokens:cn` ☆ Analogue of `\seq_map_function:NN` which maps several tokens instead of a single function. The `<code>` receives each item in the `<seq var>` as a trailing brace group. For instance,

`\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }`

expands to twice each item in the `<seq var>`: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives 2 and `<item>` as its two arguments. The function `\seq_map_inline:Nn` is typically faster but it is not expandable.

`\seq_map_variable:NNn` `\seq_map_variable:NNn <seq var> <variable> {<code>}`
`\seq_map_variable:(Ncn|cNn|ccn)`

Stores each `<item>` of the `<seq var>` in turn in the (token list) `<variable>` and applies the `<code>`. The `<code>` will usually make use of the `<variable>`, but this is not enforced. The assignments to the `<variable>` are local. Its value after the loop is the last `<item>` in the `<seq var>`, or its original value if the `<seq var>` is empty. The `<items>` are returned from left to right.

`\seq_map_indexed_function:NN` ☆ `\seq_map_indexed_function:NN <seq var> <function>`

Applies `<function>` to every entry in the `<seq var>`. The `<function>` should have signature `:nn`. It receives two arguments for each iteration: the `<index>` (namely 1 for the first entry, then 2 and so on) and the `<item>`.

`\seq_map_indexed_inline:Nn` `\seq_map_indexed_inline:Nn <seq var> {<inline function>}`

Applies `<inline function>` to every entry in the `<seq var>`. The `<inline function>` should consist of code which receives the `<index>` (namely 1 for the first entry, then 2 and so on) as `#1` and the `<item>` as `#2`.

`\seq_map_pairwise_function:NNN` ☆ `\seq_map_pairwise_function:NNN <seq var12
\seq_map_pairwise_function:(Ncn|cNn|ccN) ☆`

New: 2023-05-10

Applies `<function>` to every pair of items `<seq var1–<seq var2 from the two sequences, returning items from both sequences from left to right. The <function> receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (i.e., whichever sequence has fewer items determines how many iterations occur).`

<code>\seq_map_break: ☆ \seq_map_break:</code>	Used to terminate a <code>\seq_map...</code> function before all entries in the <code>⟨seq var⟩</code> have been processed. This normally takes place within a conditional statement, for example <pre> \seq_map_inline:Nn \l_my_seq { \str_if_eq:nnTF { #1 } { bingo } { \seq_map_break: } { % Do something useful } } </pre> Use outside of a <code>\seq_map...</code> scenario leads to low level T_EX errors. T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

<code>\seq_map_break:n ☆ \seq_map_break:n {⟨code⟩}</code>	Used to terminate a <code>\seq_map...</code> function before all entries in the <code>⟨seq var⟩</code> have been processed, inserting the <code>⟨code⟩</code> after the mapping has ended. This normally takes place within a conditional statement, for example <pre> \seq_map_inline:Nn \l_my_seq { \str_if_eq:nnTF { #1 } { bingo } { \seq_map_break:n { <code> } } { % Do something useful } } </pre> Use outside of a <code>\seq_map...</code> scenario leads to low level T_EX errors. T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the <code>⟨code⟩</code> is inserted into the input stream. This depends on the design of the mapping function.

<code>\seq_set_map:NNn \seq_set_map:NNn ⟨seq var₁⟩ ⟨seq var₂⟩ {⟨inline function⟩}</code>	
<code>\seq_gset_map:NNn</code>	Applies <code>⟨inline function⟩</code> to every <code>⟨item⟩</code> stored within the <code>⟨seq var₂⟩</code> . The <code>⟨inline function⟩</code> should consist of code which will receive the <code>⟨item⟩</code> as <code>#1</code> . The sequence resulting from applying <code>⟨inline function⟩</code> to each <code>⟨item⟩</code> is assigned to <code>⟨seq var₁⟩</code> .

Updated: 2020-07-16

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

<code>\seq_set_map_e:NNn</code>	<code>\seq_set_map_e:NNn <seq var₁> <seq var₂> {(inline function)}</code>
<code>\seq_gset_map_e:NNn</code>	Applies <i><inline function></i> to every <i><item></i> stored within the <i><seq var₂></i> . The <i><inline function></i> should consist of code which will receive the <i><item></i> as #1. The sequence resulting from e-expanding <i><inline function></i> applied to each <i><item></i> is assigned to <i><seq var₁></i> . As such, the code in <i><inline function></i> should be expandable.

New: 2020-07-16

Updated: 2023-10-26

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level T_EX errors.

<code>\seq_count:N *</code>	<code>\seq_count:N <seq var></code>
<code>\seq_count:c *</code>	Leaves the number of items in the <i><seq var></i> in the input stream as an <i><integer denotation></i> . The total number of items in a <i><seq var></i> includes those which are empty and duplicates, i.e., every item in a <i><seq var></i> is unique.

21.8 Using the content of sequences directly

<code>\seq_use:Nnnn *</code>	<code>\seq_use:Nnnn <seq var> {(separator between two)}</code>
<code>\seq_use:cnnn *</code>	<code>{(separator between more than two)} {(separator between final two)}</code>

Places the contents of the *<seq var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the sequence has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the sequence has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an e-type or x-type argument expansion.

```
\seq_use:Nn * \seq_use:Nn <seq var> {<separator>}
```

```
\seq_use:cn *
```

Places the contents of the $\langle seq\ var \rangle$ in the input stream, with the $\langle separator \rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator \rangle$, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TpXhackers note: The result is returned within the $\backslash unexpanded$ primitive ($\backslash exp_not:n$), which means that the $\langle items \rangle$ do not expand further when appearing in an **e**-type or **x**-type argument expansion.

```
\seq_format:Nn * \seq_format:Nn <seq var> {<format specification>}
```

```
\seq_format:cn *
```

Converts each item in the $\langle seq\ var \rangle$ as a token list to a string according to the $\langle format\ specification \rangle$, and concatenates the results. The details of the $\langle format\ specification \rangle$ are described in Section 19.1.

New: 2025-06-09

21.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

```
\seq_get:NN \seq_get:NN <seq var> <t1 var>
```

```
\seq_get:cn
```

Reads the top item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$ without removing it from the $\langle seq\ var \rangle$. The $\langle t1\ var \rangle$ is assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker $\backslash q\_no\_value$.

```
\seq_pop:NN \seq_pop:NN <seq var> <t1 var>
```

```
\seq_pop:cn
```

Pops the top item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$. Both of the variables are assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker $\backslash q\_no\_value$.

```
\seq_gpop:NN \seq_gpop:NN <seq var> <t1 var>
```

```
\seq_gpop:cn
```

Pops the top item from a $\langle seq\ var \rangle$ into the $\langle t1\ var \rangle$. The $\langle seq\ var \rangle$ is modified globally, while the $\langle t1\ var \rangle$ is assigned locally. If $\langle seq\ var \rangle$ is empty the $\langle t1\ var \rangle$ is set to the special marker $\backslash q\_no\_value$.

```
\seq_get:NNTF \seq_get:NNTF <seq var> <t1 var> {<true code>} {<false code>}
```

```
\seq_get:cnTF
```

If the $\langle seq\ var \rangle$ is empty, leaves the $\langle false\ code \rangle$ in the input stream. The value of the $\langle t1\ var \rangle$ is not defined in this case and should not be relied upon. If the $\langle seq\ var \rangle$ is non-empty, stores the top item from a $\langle seq\ var \rangle$ in the $\langle t1\ var \rangle$ without removing it from the $\langle seq\ var \rangle$. The $\langle t1\ var \rangle$ is assigned locally.

`\seq_pop:NNTF` `\seq_pop:NNTF <seq var> <tl var> {<true code>} {<false code>}`

`\seq_pop:cNNTF` If the `<seq var>` is empty, leaves the `<false code>` in the input stream. The value of the `<tl var>` is not defined in this case and should not be relied upon. If the `<seq var>` is non-empty, pops the top item from the `<seq var>` in the `<tl var>`, i.e., removes the item from the `<seq var>`. Both the `<seq var>` and the `<tl var>` are assigned locally.

`\seq_gpop:NNTF` `\seq_gpop:NNTF <seq var> <tl var> {<true code>} {<false code>}`

`\seq_gpop:cNNTF` If the `<seq var>` is empty, leaves the `<false code>` in the input stream. The value of the `<tl var>` is not defined in this case and should not be relied upon. If the `<seq var>` is non-empty, pops the top item from the `<seq var>` in the `<tl var>`, i.e., removes the item from the `<seq var>`. The `<seq var>` is modified globally, while the `<tl var>` is assigned locally.

`\seq_push:Nn` `\seq_push:Nn <seq var> {<item>}`

`\seq_push:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`

`\seq_gpush:Nn`

`\seq_gpush:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`

Adds the `{<item>}` to the top of the `<seq var>`.

21.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a `<seq var>` only has distinct items, use `\seq_remove_duplicates:N <seq var>`. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set `<seq var>` are straightforward. For instance, `\seq_count:N <seq var>` expands to the number of items, while `\seq_if_in:NnTF <seq var> {<item>}` tests if the `<item>` is in the set.

Adding an `<item>` to a set `<seq var>` can be done by appending it to the `<seq var>` if it is not already in the `<seq var>`:

```
\seq_if_in:NnF <seq var> {<item>}
{ \seq_put_right:Nn <seq var> {<item>} }
```

Removing an `<item>` from a set `<seq var>` can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn <seq var> {<item>}
```

The intersection of two sets `<seq var1>` and `<seq var2>` can be stored into `<seq var3>` by collecting items of `<seq var1>` which are in `<seq var2>`.

```

\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
  \seq_if_in:NnT <seq var2> {#1}
  { \seq_put_right:Nn <seq var3> {#1} }
}

```

The code as written here only works if $\langle \text{seq var}_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l\_ \langle \text{pkg} \rangle\_ \text{tmp\_seq}$, then $\langle \text{seq var}_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle \text{seq var}_1 \rangle$ one by one

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
  \seq_if_in:NnF <seq var3> {#1}
  { \seq_put_right:Nn <seq var3> {#1} }
}

```

The second approach is faster than the first when the $\langle \text{seq var}_2 \rangle$ is short compared to $\langle \text{seq var}_1 \rangle$.

The difference of two sets $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ by removing items of the $\langle \text{seq var}_2 \rangle$ from (a copy of) the $\langle \text{seq var}_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }

```

The symmetric difference of two sets $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ can be stored into $\langle \text{seq var}_3 \rangle$ by computing the difference between $\langle \text{seq var}_1 \rangle$ and $\langle \text{seq var}_2 \rangle$ and storing the result as $\backslash l\_ \langle \text{pkg} \rangle\_ \text{tmp\_seq}$, then the difference between $\langle \text{seq var}_2 \rangle$ and $\langle \text{seq var}_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle \text{pkg} \rangle\_ \text{tmp\_seq} <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l\_ \langle \text{pkg} \rangle\_ \text{tmp\_seq} {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l\_ \langle \text{pkg} \rangle\_ \text{tmp\_seq}

```

21.11 Constant and scratch sequences

$\backslash \text{c\_empty\_seq}$ Constant that is always empty.

<code>\l_tmpa_seq</code>	Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_seq</code>	

<code>\g_tmpa_seq</code>	Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_seq</code>	

21.12 Viewing sequences

<code>\seq_show:N</code>	<code>\seq_show:N</code> $\langle seq\ var \rangle$
<code>\seq_show:c</code>	Displays the entries in the $\langle seq\ var \rangle$ in the terminal.

Updated: 2021-04-29

<code>\seq_log:N</code>	<code>\seq_log:N</code> $\langle seq\ var \rangle$
<code>\seq_log:c</code>	Writes the entries in the $\langle seq\ var \rangle$ in the log file.

Updated: 2021-04-29

Chapter 22

The l3int module

Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`<int expr>`”).

22.1 Integer expressions

Throughout this module, (almost) all `n`-type argument allow for an `<intexpr>` argument with the following syntax. The `<integer expression>` should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large (but the operands a , b , c are still constrained to an absolute value at most $2^{31} - 1$);
- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl  
\tl_set:Nn \l_my_tl { 5 }  
\int_new:N \l_my_int  
\int_set:Nn \l_my_int { 4 }  
\int_show:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result -6 because `\l_my_tl` expands to the integer denotation 5 while the integer variable `\l_my_int` takes the value 4. As the *integer expression* is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_not:N` may incorrectly interrupt the expression.

`\int_eval:n` * `\int_eval:n {<int expr>}`

Evaluates the *int expr* and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results $-$ followed by such a sequence, and 0 for zero.

T_EXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *internal integer*, and therefore requires suitable termination if used in a T_EX-style integer assignment.

As all T_EX integers, integer operands can also be dimension or skip variables, converted to integers in **sp**, or octal numbers given as `'` followed by digits other than 8 and 9, or hexadecimal numbers given as `"` followed by digits or upper case letters from **A** to **F**, or the character code of some character or one-character control sequence, given as `'<char>`.

`\int_eval:w` * `\int_eval:w <int expr>`

Evaluates the *int expr* as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop:` it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1_+1_9` (with explicit space tokens inserted using `~` in a code setting) expands to 29 since the digit 9 is not part of the expression. Expansion details, etc., are as given for `\int_eval:n`.

`\int_sign:n` * `\int_sign:n {<int expr>}`

Evaluates the *int expr* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\int_abs:n` * `\int_abs:n {<int expr>}`

Evaluates the *int expr* as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an *integer denotation* after two expansions.

`\int_div_round:nn` * `\int_div_round:nn {<int expr1>} {<int expr2>}`

Evaluates the two *int expr*s as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an *int expr*. The result is left in the input stream as an *integer denotation* after two expansions.

`\int_div_truncate:nn` * `\int_div_truncate:nn {<int expr1>} {<int expr2>}`

Evaluates the two *int expr*s as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an *integer denotation* after two expansions.

<code>\int_max:nn</code>	<code>★ \int_max:nn {\int expr_1} {\int expr_2}</code>
<code>\int_min:nn</code>	<code>★ \int_min:nn {\int expr_1} {\int expr_2}</code>

Evaluates the $\langle \text{int expr} \rangle$ s as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an $\langle \text{integer denotation} \rangle$ after two expansions.

<code>\int_mod:nn</code>	<code>★ \int_mod:nn {\int expr_1} {\int expr_2}</code>

Evaluates the two $\langle \text{int expr} \rangle$ s as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div-truncate:nn` $\{\langle \text{int expr}_1 \rangle\} \{\langle \text{int expr}_2 \rangle\}$ times $\langle \text{int expr}_2 \rangle$ from $\langle \text{int expr}_1 \rangle$. Thus, the result has the same sign as $\langle \text{int expr}_1 \rangle$ and its absolute value is strictly less than that of $\langle \text{int expr}_2 \rangle$. The result is left in the input stream as an $\langle \text{integer denotation} \rangle$ after two expansions.

22.2 Creating and initializing integers

<code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<code>\int_new:c</code>	

Creates a new $\langle \text{integer} \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle \text{integer} \rangle$ is initially equal to 0.

<code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\int expr}</code>
<code>\int_const:cn</code>	

Creates a new constant $\langle \text{integer} \rangle$ or raises an error if the name is already taken. The value of the $\langle \text{integer} \rangle$ is set globally to the $\langle \text{int expr} \rangle$.

<code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<code>\int_zero:c</code>	
<code>\int_gzero:N</code>	Sets $\langle \text{integer} \rangle$ to 0.
<code>\int_gzero:c</code>	

<code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>

Ensures that the $\langle \text{integer} \rangle$ exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the $\langle \text{integer} \rangle$ set to zero.

<code>\int_set_eq:NN</code>	<code>\int_set_eq:NN \langle integer_1 \rangle \langle integer_2 \rangle</code>
<code>\int_set_eq:(cN Nc cc)</code>	
<code>\int_gset_eq:NN</code>	Sets the content of $\langle \text{integer}_1 \rangle$ equal to that of $\langle \text{integer}_2 \rangle$.
<code>\int_gset_eq:(cN Nc cc)</code>	

<code>\int_if_exist_p:N</code>	<code>★ \int_if_exist_p:N \langle integer \rangle</code>
<code>\int_if_exist_p:c</code>	<code>★ \int_if_exist:NTF \langle integer \rangle {\langle true code \rangle} {\langle false code \rangle}</code>

<code>\int_if_exist:NTF</code>	<code>★</code>	Tests whether the $\langle \text{integer} \rangle$ is currently defined. This does not check that the $\langle \text{integer} \rangle$ really is an integer variable.
<code>\int_if_exist:cTF</code>	<code>★</code>	

22.3 Setting and incrementing integers

<code>\int_add:Nn</code>	<code>\int_add:Nn <integer> {<int expr>}</code>
<code>\int_add:cn</code>	
<code>\int_gadd:Nn</code>	Adds the result of the <code><int expr></code> to the current content of the <code><integer></code> .
<code>\int_gadd:cn</code>	

<code>\int_decr:N</code>	<code>\int_decr:N <integer></code>
<code>\int_decr:c</code>	
<code>\int_gdecr:N</code>	Decreases the value stored in <code><integer></code> by 1.
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N <integer></code>
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in <code><integer></code> by 1.
<code>\int_gincr:c</code>	

<code>\int_set:Nn</code>	<code>\int_set:Nn <integer> {<int expr>}</code>
<code>\int_set:(cn NV cV)</code>	
<code>\int_gset:Nn</code>	Sets <code><integer></code> to the value of <code><int expr></code> , which must evaluate to an integer (as described for <code>\int_eval:n</code>).
<code>\int_gset:(cn NV cV)</code>	

<code>\int_set_regex_count:Nnn</code>	<code>\int_set_regex_count:Nnn <integer> {<regex>} {<token list>}</code>
<code>\int_set_regex_count:cnn</code>	<code>\int_set_regex_count:Nnn <integer> <regex var> {<token list>}</code>
<code>\int_set_regex_count:NNn</code>	
<code>\int_set_regex_count:cNn</code>	Sets <code><integer></code> equal to the number of times <code><regular expression></code> appears in <code><token list></code> . The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,
<code>\int_gset_regex_count:Nnn</code>	
<code>\int_gset_regex_count:cnn</code>	
<code>\int_gset_regex_count:NNn</code>	
<code>\int_gset_regex_count:cNn</code>	

New: 2024-12-08

`\int_set_regex_count:Nnn \l_foo_int { (b+|c) } { abbababcbb }`

results in `\l_foo_int` taking the value 5. Theses are alternative names for `\regex_count:nnN` and friends, with arguments re-ordered for `<integer>` setting; see `l3regex` chapter for more details of the `<regex>` format.

<code>\int_sub:Nn</code>	<code>\int_sub:Nn <integer> {<int expr>}</code>
<code>\int_sub:cn</code>	
<code>\int_gsub:Nn</code>	Subtracts the result of the <code><int expr></code> from the current content of the <code><integer></code> .
<code>\int_gsub:cn</code>	

22.4 Using integers

<code>\int_use:N</code>	<code>*</code>	<code>\int_use:N</code>	<code><integer></code>
<code>\int_use:c</code>	<code>*</code>	Recovers the content of an <code><integer></code> and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an <code><integer></code> is required (such as in the first and third arguments of <code>\int_compare:nNnTF</code>).	

T_EXhackers note: `\int_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

22.5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	<code>*</code>	<code>\int_compare_p:nNn</code>	<code>{<int expr₁>} <relation> {<int expr₂>}</code>
<code>\int_compare_p:(vNn nNv vNv)</code>	<code>*</code>	<code>\int_compare:nNnTF</code>	
<code>\int_compare:nNnTF</code>	<code>*</code>	<code>{<int expr₁>} <relation> {<int expr₂>}</code>	
<code>\int_compare:(vNn nNv vNv)TF</code>	<code>*</code>	<code>{<true code>} {<false code>}</code>	

This function first evaluates each of the `<int expr>`s as described for `\int_eval:n`. The two results are then compared using the `<relation>`:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

Variables may be used directly in the `<int exprs>` here: as a result, there is no need for V- (or e-) type variants. The v-type variants are provided for convenience.

```

\int_compare_p:n * \int_compare_p:n
\int_compare:nTF * {
    <int expr1> <relation1>
    ...
    <int exprN> <relationN>
    <int exprN+1>
}
\int_compare:nTF
{
    <int expr1> <relation1>
    ...
    <int exprN> <relationN>
    <int exprN+1>
}
{<true code>} {<false code>}

```

This function evaluates the `<int expr>`s as described for `\int_eval:n` and compares consecutive result using the corresponding `<relation>`, namely it compares `<int expr1>` and `<int expr2>` using the `<relation1>`, then `<int expr2>` and `<int expr3>` using the `<relation2>`, until finally comparing `<int exprN>` and `<int exprN+1>` using the `<relationN>`. The test yields `true` if all comparisons are `true`. Each `<int expr>` is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other `<integer expression>` is evaluated and no other comparison is performed. The `<relations>` can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

```

\int_case:nn * \int_case:nnTF {\test int expr}
\int_case:nnTF * {
    {\int expr case1} {\code case1}
    {\int expr case2} {\code case2}
    ...
    {\int expr casen} {\code casen}
}
{\true code}
{\false code}

```

This function evaluates the $\langle \text{test int expr} \rangle$ and compares this in turn to each of the $\langle \text{int expr case} \rangle$ s until a match is found. If the two are equal then the associated $\langle \text{code} \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle \text{true code} \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle \text{false code} \rangle$ is inserted. The function $\backslash\text{int\_case:nn}$, which does nothing if there is no match, is also available. For example

```

\int_case:nnF
{ 2 * 5 }
{
    { 5 }      { Small }
    { 4 + 6 }  { Medium }
    { -2 * 10 } { Negative }
}
{ No idea! }

```

leaves “Medium” in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

```

\int_if_even_p:n * \int_if_odd_p:n {\int expr}
\int_if_even:nTF * \int_if_odd:nTF {\int expr}
\int_if_odd_p:n * {\true code} {\false code}
\int_if_odd:nTF *

```

This function first evaluates the $\langle \text{int expr} \rangle$ as described for $\backslash\text{int\_eval:n}$. It then evaluates if this is odd or even, as appropriate.

```

\int_if_zero_p:n * \int_if_zero_p:n {\int expr}
\int_if_zero:nTF * \int_if_zero:nTF {\int expr}
{\true code} {\false code}

```

New: 2023-05-17

This function first evaluates the $\langle \text{int expr} \rangle$ as described for $\backslash\text{int\_eval:n}$. It then evaluates if this is zero or not.

22.6 Integer expression loops

```

\int_do_until:nNnn ☆ \int_do_until:nNnn {\int expr1} <relation> {\int expr2} {\code}

```

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle \text{int expr} \rangle$ s as described for $\backslash\text{int\_compare:nNnTF}$. If the test is **false** then the $\langle \text{code} \rangle$ is inserted into the input stream again and a loop occurs until the $\langle \text{relation} \rangle$ is **true**.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while:nNnn {<int expr₁>} <relation> {<int expr₂>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\int_until_do:nNnn</code> ☆	<code>\int_until_do:nNnn {<int expr₁>} <relation> {<int expr₂>} {<code>}</code>
	Evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nNnn</code> ☆	<code>\int_while_do:nNnn {<int expr₁>} <relation> {<int expr₂>} {<code>}</code>
	Evaluates the relationship between the two <code><int expr></code> s as described for <code>\int_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\int_do_until:nn</code> ☆	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> . If the test is <code>false</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>true</code> .
<code>\int_do_while:nn</code> ☆	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\int_until_do:nn</code> ☆	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
	Evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nn</code> ☆	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
	Evaluates the <code><integer relation></code> as described for <code>\int_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .

22.7 Integer step functions

<code>\int_step_function:nN</code>	☆	<code>\int_step_function:nN {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnN</code>	☆	<code>\int_step_function:nnN {⟨initial value⟩} {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnnN</code>	☆	<code>\int_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩</code>

This function first evaluates the `⟨initial value⟩`, `⟨step⟩` and `⟨final value⟩`, all of which should be integer expressions. The `⟨function⟩` is then placed in front of each `⟨value⟩` from the `⟨initial value⟩` to the `⟨final value⟩` in turn (using `⟨step⟩` between each `⟨value⟩`). The `⟨step⟩` must be non-zero. If the `⟨step⟩` is positive, the loop stops when the `⟨value⟩` becomes larger than the `⟨final value⟩`. If the `⟨step⟩` is negative, the loop stops when the `⟨value⟩` becomes smaller than the `⟨final value⟩`. The `⟨function⟩` should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

The functions `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed `⟨step⟩` of 1, and in the case of `\int_step_function:nN` the `⟨initial value⟩` is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_tokens:nn</code>	☆	<code>\int_step_tokens:nn {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_tokens:nnn</code>	☆	<code>\int_step_tokens:nnn {⟨initial value⟩} {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_tokens:nnnn</code>	☆	<code>\int_step_tokens:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

New: 2025-01-13

This function works just like `\int_step_function:nnnN` but instead of mapping a single function to each stepped `⟨value⟩` between `⟨initial value⟩` and `⟨final value⟩` this maps the multiple tokens in `⟨code⟩`, so that it gets the current `⟨value⟩` as a braced argument following it. For instance

```
\cs_set:Npn \my_product:nn #1#2
{ $ #1 \times #2 = \int_eval:n { #1 * #2 }$ \quad }
\int_step_tokens:nnnn { 1 } { 1 } { 4 } { \my_product:nn { 2 } }
```

would print

```
2 × 1 = 2 2 × 2 = 4 2 × 3 = 6 2 × 4 = 8
```

<code>\int_step_inline:nn</code>	<code>\int_step_inline:nn {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnn</code>	<code>\int_step_inline:nnn {⟨initial value⟩} {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnnn</code>	<code>\int_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

This function first evaluates the `⟨initial value⟩`, `⟨step⟩` and `⟨final value⟩`, all of which should be integer expressions. Then for each `⟨value⟩` from the `⟨initial value⟩` to the `⟨final value⟩` in turn (using `⟨step⟩` between each `⟨value⟩`), the `⟨code⟩` is inserted into the input stream with `#1` replaced by the current `⟨value⟩`. Thus the `⟨code⟩` should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed `⟨step⟩` of 1, and in the case of `\int_step_inline:nn` the `⟨initial value⟩` is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_variable:nNn</code>	<code>\int_step_variable:nNn {<final value>} <tl var> {<code>}</code>
<code>\int_step_variable:nnNn</code>	<code>\int_step_variable:nnNn {<initial value>} {<final value>} <tl var> {<code>}</code>
<code>\int_step_variable:nnnNn</code>	<code>\int_step_variable:nnnNn {<initial value>} {<step>} {<final value>} <tl var> {<code>}</code>

This function first evaluates the `<initial value>`, `<step>` and `<final value>`, all of which should be integer expressions. Then for each `<value>` from the `<initial value>` to the `<final value>` in turn (using `<step>` between each `<value>`), the `<code>` is inserted into the input stream, with the `<tl var>` defined as the current `<value>`. Thus the `<code>` should make use of the `<tl var>`.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed `<step>` of 1, and in the case of `\int_step_variable:nNn` the `<initial value>` is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

22.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

<code>\int_to_arabic:n *</code>	<code>\int_to_arabic:n {<int expr>}</code>
<code>\int_to_arabic:v *</code>	Places the value of the <code><int expr></code> in the input stream as digits, with category code 12 (other).

<code>\int_to_alph:n *</code>	<code>\int_to_alph:n {<int expr>}</code>
<code>\int_to_Alph:n *</code>	Evaluates the <code><int expr></code> and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

`\int_to_alph:n { 1 }`

places **a** in the input stream,

`\int_to_alph:n { 26 }`

is represented as **z** and

`\int_to_alph:n { 27 }`

is converted to **aa**. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

```
\int_to_symbols:nnn * \int_to_symbols:nnn
                        {\int expr} {\total symbols}
                        {\value to symbol mapping}
```

This is the low-level function for conversion of an $\langle \text{int expr} \rangle$ into a symbolic form (often letters). The $\langle \text{total symbols} \rangle$ available should be given as an integer expression. Values are actually converted to symbols according to the $\langle \text{value to symbol mapping} \rangle$. This should be given as $\langle \text{total symbols} \rangle$ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

```
\int_to_bin:n * \int_to_bin:n {\int expr}
```

Calculates the value of the $\langle \text{int expr} \rangle$ and places the binary representation of the result in the input stream.

```
\int_to_hex:n * \int_to_hex:n {\int expr}
```

```
\int_to_Hex:n * 
```

Calculates the value of the $\langle \text{int expr} \rangle$ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

```
\int_to_oct:n * \int_to_oct:n {\int expr}
```

Calculates the value of the $\langle \text{int expr} \rangle$ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

```
\int_to_base:nn * \int_to_base:nn {\int expr} {\base}
```

```
\int_to_Base:nn * 
```

Calculates the value of the $\langle \text{int expr} \rangle$ and converts it into the appropriate representation in the $\langle \text{base} \rangle$; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum $\langle \text{base} \rangle$ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

T_EXhackers note: This is a generic version of `\int_to_bin:n`, etc.

<code>\int_to_roman:n</code> ☆	<code>\int_to_roman:n {⟨int expr⟩}</code>
<code>\int_to_Roman:n</code> ☆	Places the value of the <code>⟨int expr⟩</code> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are <code>mdclxvi</code> , repeated as needed: the notation with bars (such as <code>̄v</code> for 5000) is <i>not</i> used. For instance <code>\int_to_roman:n { 8249 }</code> expands to <code>mmmmmmmmccxlix</code> .
<code>\int_format:nn</code> ☆	<code>\int_format:nn {⟨int expr⟩} {⟨format specification⟩}</code>
New: 2025-06-09	Evaluates the <code>⟨int expr⟩</code> and converts the result to a string according to the <code>⟨format specification⟩</code> . The <code>⟨precision⟩</code> argument is not allowed. The <code>⟨style⟩</code> can be <code>b</code> for binary output, <code>d</code> for decimal output (this is the default), <code>o</code> for octal output, <code>X</code> for hexadecimal output (using capital letters). The details of the <code>⟨format specification⟩</code> are described in Section 19.1 .

22.9 Converting from other formats to integers

<code>\int_from_alph:n</code> ☆	<code>\int_from_alph:n {⟨letters⟩}</code>
	Converts the <code>⟨letters⟩</code> into the integer (base 10) representation and leaves this in the input stream. The <code>⟨letters⟩</code> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of <code>+</code> and <code>-</code> . This is the inverse function of <code>\int_to_alph:n</code> and <code>\int_to_Alph:n</code> .
<code>\int_from_bin:n</code> ☆	<code>\int_from_bin:n {⟨binary number⟩}</code>
	Converts the <code>⟨binary number⟩</code> into the integer (base 10) representation and leaves this in the input stream. The <code>⟨binary number⟩</code> is first converted to a string, with no expansion. The function accepts a leading sign, made of <code>+</code> and <code>-</code> , followed by binary digits. This is the inverse function of <code>\int_to_bin:n</code> .
<code>\int_from_hex:n</code> ☆	<code>\int_from_hex:n {⟨hexadecimal number⟩}</code>
	Converts the <code>⟨hexadecimal number⟩</code> into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the <code>⟨hexadecimal number⟩</code> by upper or lower case letters. The <code>⟨hexadecimal number⟩</code> is first converted to a string, with no expansion. The function also accepts a leading sign, made of <code>+</code> and <code>-</code> . This is the inverse function of <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code> .
<code>\int_from_oct:n</code> ☆	<code>\int_from_oct:n {⟨octal number⟩}</code>
	Converts the <code>⟨octal number⟩</code> into the integer (base 10) representation and leaves this in the input stream. The <code>⟨octal number⟩</code> is first converted to a string, with no expansion. The function accepts a leading sign, made of <code>+</code> and <code>-</code> , followed by octal digits. This is the inverse function of <code>\int_to_oct:n</code> .

<code>\int_from_roman:n</code>	<code>*</code>	<code>\int_from_roman:n {⟨roman numeral⟩}</code>

Converts the $\langle roman numeral \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman numeral \rangle$ is first converted to a string, with no expansion. The $\langle roman numeral \rangle$ may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value is -1 . This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

<code>\int_from_base:nn</code>	<code>*</code>	<code>\int_from_base:nn {⟨number⟩} {⟨base⟩}</code>

Converts the $\langle number \rangle$ expressed in $\langle base \rangle$ into the appropriate value in base 10. The $\langle number \rangle$ is first converted to a string, with no expansion. The $\langle number \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base \rangle$ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

22.10 Random integers

<code>\int_rand:nn</code>	<code>*</code>	<code>\int_rand:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>

Evaluates the two $\langle int expr \rangle$ s and produces a pseudo-random number between the two (with bounds included).

<code>\int_rand:n</code>	<code>*</code>	<code>\int_rand:n {⟨int expr⟩}</code>

Evaluates the $\langle int expr \rangle$ then produces a pseudo-random number between 1 and the $\langle int expr \rangle$ (included).

22.11 Viewing integers

<code>\int_show:N</code>	<code>\int_show:N ⟨integer⟩</code>

<code>\int_show:c</code>	Displays the value of the $\langle integer \rangle$ on the terminal.

<code>\int_show:n</code>	<code>\int_show:n {⟨int expr⟩}</code>

Displays the result of evaluating the $\langle int expr \rangle$ on the terminal.

<code>\int_log:N</code>	<code>\int_log:N ⟨integer⟩</code>

<code>\int_log:c</code>	Writes the value of the $\langle integer \rangle$ in the log file.

<code>\int_log:n</code>	<code>\int_log:n {⟨int expr⟩}</code>

Writes the result of evaluating the $\langle int expr \rangle$ in the log file.

22.12 Constant integers

<u><code>\c_zero_int</code></u>	Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.
<u><code>\c_one_int</code></u>	

<u><code>\c_max_int</code></u>	The maximum value that can be stored as an integer.

<u><code>\c_max_register_int</code></u>	Maximum number of registers.

<u><code>\c_max_char_int</code></u>	Maximum character code completely supported by the engine.

22.13 Scratch integers

<u><code>\l_tmpa_int</code></u>	Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u><code>\l_tmpb_int</code></u>	

<u><code>\g_tmpa_int</code></u>	Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u><code>\g_tmpb_int</code></u>	

22.14 Direct number expansion

`\int_value:w` ★ `\int_value:w` $\langle integer \rangle$
`\int_value:w` $\langle integer\ denotation \rangle$ $\langle optional\ space \rangle$

Expands the following tokens until an $\langle integer \rangle$ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The $\langle integer \rangle$ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any T_EX register except `\toks`) or
- explicit digits (or by ‘ $\langle octal\ digits \rangle$ ’ or “ $\langle hexadecimal\ digits \rangle$ ” or ‘ $\langle character \rangle$ ’).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, `\int_eval:n` is the preferred approach to generating numbers.

T_EXhackers note: This is the T_EX primitive `\number`.

22.15 Primitive conditionals

`\if_int_compare:w` ★ `\if_int_compare:w` $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifnum`.

`\if_case:w` ★ `\if_case:w` $\langle integer \rangle$ $\langle case_0 \rangle$
`\or:` ★ `\or:` $\langle case_1 \rangle$
`\or:` ...
`\else:` $\langle default \rangle$
`\fi:`

Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, etc. The $\langle integer \rangle$ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

`\if_int_odd:w` ★ `\if_int_odd:w` $\langle tokens \rangle$ $\langle optional\ space \rangle$
 $\langle true\ code \rangle$
 `\else:`
 $\langle true\ code \rangle$
 `\fi:`

Expands $\langle tokens \rangle$ until a non-numeric token or a space is found, and tests whether the resulting $\langle integer \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifodd`.

Chapter 23

The l3flag module

Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any (small) non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height and that the memory cannot be reclaimed even if the flag is cleared. Flags should not be used unless it is unavoidable.

In earlier versions, flags were referenced by an n-type *flag name* such as `fp_overflow`, used as part of `\use:c` constructions. All of the commands described below have n-type analogues that can still appear in old code, but the N-type commands are to be preferred moving forward. The n-type *flag name* is simply mapped to `\l_<flag name>_flag`, which makes it easier for packages using public flags (such as `l3fp`) to retain backwards compatibility.

23.1 Setting up flags

<code>\flag_new:N</code>	<code>\flag_new:N <flag var></code>

<code>\flag_new:c</code>	

<code>New: 2024-01-12</code>	Creates a new <i>flag var</i> , or raises an error if the name is already taken. The declaration is global, but flags are always local variables. The <i>flag var</i> initially has zero height.

<code>\flag_clear:N</code>	<code>\flag_clear:N</code> $\langle flag\ var \rangle$
<code>\flag_clear:c</code>	Sets the height of the $\langle flag\ var \rangle$ to zero. The assignment is local.
New: 2024-01-12	

<code>\flag_clear_new:N</code>	<code>\flag_clear_new:N</code> $\langle flag\ var \rangle$
<code>\flag_clear_new:c</code>	Ensures that the $\langle flag\ var \rangle$ exists globally by applying <code>\flag_new:N</code> if necessary, then applies <code>\flag_clear:N</code> , setting the height to zero locally.
New: 2024-01-12	

<code>\flag_show:N</code>	<code>\flag_show:N</code> $\langle flag\ var \rangle$
<code>\flag_show:c</code>	Displays the height of the $\langle flag\ var \rangle$ in the terminal.
New: 2024-01-12	

<code>\flag_log:N</code>	<code>\flag_log:N</code> $\langle flag\ var \rangle$
<code>\flag_log:c</code>	Writes the height of the $\langle flag\ var \rangle$ in the log file.
New: 2024-01-12	

23.2 Expandable flag commands

<code>\flag_if_exist_p:N</code>	<code>\flag_if_exist_p:N</code> $\langle flag\ var \rangle$
<code>\flag_if_exist_p:c</code>	<code>\flag_if_exist:NTF</code> $\langle flag\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\flag_if_exist:NTF</code>	$\star$ This function returns <code>true</code> if the $\langle flag\ var \rangle$ is currently defined, and <code>false</code> otherwise.
<code>\flag_if_exist:cTF</code>	$\star$ This does not check that the $\langle flag\ var \rangle$ really is a flag variable.
New: 2024-01-12	

<code>\flag_if_raised_p:N</code>	<code>\flag_if_raised_p:N</code> $\langle flag\ var \rangle$
<code>\flag_if_raised_p:c</code>	<code>\flag_if_raised:NTF</code> $\langle flag\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\flag_if_raised:NTF</code>	$\star$ This function returns <code>true</code> if the $\langle flag\ var \rangle$ has non-zero height, and <code>false</code> if the
<code>\flag_if_raised:cTF</code>	$\star$ $\langle flag\ var \rangle$ has zero height.
New: 2024-01-12	

<code>\flag_height:N</code>	<code>\flag_height:N</code> $\langle flag\ var \rangle$
<code>\flag_height:c</code>	Expands to the height of the $\langle flag\ var \rangle$ as an integer denotation.
New: 2024-01-12	

<code>\flag_raise:N</code>	<code>\flag_raise:N</code> $\langle flag\ var \rangle$
<code>\flag_raise:c</code>	$\star$ The height of $\langle flag\ var \rangle$ is increased by 1 locally.
New: 2024-01-12	

<code>\flag_ensure_raised:N</code>	<code>\flag_ensure_raised:N</code> $\langle flag\ var \rangle$
<code>\flag_ensure_raised:c</code>	$\star$ Ensures the $\langle flag\ var \rangle$ is raised by making its height at least 1, locally.
New: 2024-01-12	

<code>\l_tmpa_flag</code>	Scratch flag for local assignment. These are never used by the kernel code, and so are safe
<code>\l_tmpb_flag</code>	for use with any $\text{\LaTeX}$ 3-defined function. However, they may be overwritten by other
<code>New: 2024-01-12</code>	non-kernel code and so should only be used for short-term storage.

Chapter 24

The l3clist module

Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `l3seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with $\text{\LaTeX}2_{\epsilon}$ or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for `l3clist` functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already

occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and `\l3tl` functions such as `\tl_show:N` can be applied to them. (These functions often have `\clist` analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `\l3seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual `TeX` category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

24.1 Creating and initializing comma lists

<code>\clist_new:N</code>	<code>\clist_new:N <clist var></code>

<code>\clist_new:c</code>	

Creates a new `<clist var>` or raises an error if the name is already taken. The declaration is global. The `<clist var>` initially contains no items.

<code>\clist_const:Nn</code>	<code>\clist_const:Nn <clist var> {<comma list>}</code>

<code>\clist_const:(Ne cn ce)</code>	

Creates a new constant `<clist var>` or raises an error if the name is already taken. The value of the `<clist var>` is set globally to the `<comma list>`.

<code>\clist_clear:N</code>	<code>\clist_clear:N <clist var></code>

<code>\clist_clear:c</code>	

<code>\clist_gclear:N</code>	<code>\clist_gclear:N <clist var></code>

<code>\clist_gclear:c</code>	

Clears all items from the `<clist var>`.

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N <clist var></code>

<code>\clist_clear_new:c</code>	

<code>\clist_gclear_new:N</code>	

<code>\clist_gclear_new:c</code>	

Ensures that the `<clist var>` exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

<code>\clist_set_eq:NN</code>	<code>\clist_set_eq:NN <clist var₁₂</code>

<code>\clist_set_eq:(cN Nc cc)</code>	

<code>\clist_gset_eq:NN</code>	

<code>\clist_gset_eq:(cN Nc cc)</code>	

Sets the content of `<clist var1>` equal to that of `<clist var2>`. To set a token list variable equal to a comma list variable, use `\tl_set_eq:NN`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

<code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN <clist var> <seq var></code>

<code>\clist_set_from_seq:(cN Nc cc)</code>	

<code>\clist_gset_from_seq:NN</code>	

<code>\clist_gset_from_seq:(cN Nc cc)</code>	

Converts the data in the `<seq var>` into a `<clist var>`: the original `<seq var>` is unchanged. Items which contain either spaces or commas are surrounded by braces.

<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN <clist var₁> <clist var₂> <clist var₃></code>
<code>\clist_concat:ccc</code>	
<code>\clist_gconcat:NNN</code>	Concatenates the content of <code><clist var₂></code> and <code><clist var₃></code> together and saves the
<code>\clist_gconcat:ccc</code>	result in <code><clist var₁></code> . The items in <code><clist var₂></code> are placed at the left side of the new
	comma list.
<code>\clist_if_exist_p:N</code>	<code>\clist_if_exist_p:N <clist var></code>
<code>\clist_if_exist_p:c</code>	<code>\clist_if_exist:NTF <clist var> {<true code>} {<false code>}</code>
<code>\clist_if_exist:NTF</code>	<code>\clist_if_exist:NTF</code> Tests whether the <code><clist var></code> is currently defined. This does not check that the
<code>\clist_if_exist:cTF</code>	<code>\clist_if_exist:cTF</code> <code><clist var></code> really is a comma list.

24.2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn <clist var> {<item₁>, ..., <item_n>}</code>
<code>\clist_set:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV Nv Ne No cn cV cv ce co)</code>	

Sets `<clist var>` to contain the `<items>`, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_set:Nn <clist var> { {<tokens>} }`.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <clist var> {<item₁>, ..., <item_n>}</code>
<code>\clist_put_left:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV Nv Ne No cn cV cv ce co)</code>	

Appends the `<items>` to the left of the `<clist var>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_left:Nn <clist var> { {<tokens>} }`.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <clist var> {<item₁>, ..., <item_n>}</code>
<code>\clist_put_right:(NV Nv Ne No cn cV cv ce co)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV Nv Ne No cn cV cv ce co)</code>	

Appends the `<items>` to the right of the `<clist var>`. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some `<tokens>` as a single `<item>` even if the `<tokens>` contain commas or spaces, add a set of braces: `\clist_put_right:Nn <clist var> { {<tokens>} }`.

24.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

```
\clist_remove_duplicates:N \clist_remove_duplicates:N <clist var>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
```

Removes duplicate items from the $\langle\textit{clist var}\rangle$, leaving the left most copy of each item in the $\langle\textit{clist var}\rangle$. The $\langle\textit{item}\rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the $\langle\textit{clist var}\rangle$ and does a comparison with the $\langle\textit{items}\rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle\textit{clist var}\rangle$ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

```
\clist_remove_all:Nn \clist_remove_all:Nn <clist var> {<item>}
\clist_remove_all:(cn|NV|cV|Ne|ce)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV|Ne|ce)
```

Removes every occurrence of $\langle\textit{item}\rangle$ from the $\langle\textit{clist var}\rangle$. The $\langle\textit{item}\rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: The function may fail if the $\langle\textit{item}\rangle$ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

```
\clist_reverse:N \clist_reverse:N <clist var>
\clist_reverse:c
\clist_greverse:N Reverses the order of items stored in the <clist var>.
\clist_greverse:c
```

```
\clist_reverse:n ★ \clist_reverse:n {<comma list>}
```

Leaves the items in the $\langle\textit{comma list}\rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle\textit{comma list}\rangle$ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an e-type or x-type argument expansion.

```
\clist_sort:Nn \clist_sort:Nn <clist var> {<comparison code>}
\clist_sort:cn
\clist_gsort:Nn Sorts the items in the <clist var> according to the <comparison code>, and assigns the
\clist_gsort:cn result to <clist var>. The details of sorting comparison are described in Section 6.1.
```

24.4 Comma list conditionals

<code>\clist_if_empty_p:N</code>	★	<code>\clist_if_empty_p:N</code>	$\langle \textit{clist var} \rangle$
<code>\clist_if_empty_p:c</code>	★	<code>\clist_if_empty:N</code>	$\langle \textit{clist var} \rangle$ { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }
<code>\clist_if_empty:N</code>	★	Tests if the $\langle \textit{clist var} \rangle$ is empty (containing no items).	
<code>\clist_if_empty:c</code>	★		

<code>\clist_if_empty_p:n</code>	★	<code>\clist_if_empty_p:n</code>	{ $\langle \textit{comma list} \rangle$ }
<code>\clist_if_empty:n</code>	★	<code>\clist_if_empty:n</code>	{ $\langle \textit{comma list} \rangle$ } { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }

Tests if the $\langle \textit{comma list} \rangle$ is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list {~,~,~} (without outer braces) is empty, while {~,{}}, {~} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

<code>\clist_if_in:N</code>	$\langle \textit{clist var} \rangle$	<code>\clist_if_in:N</code>	$\langle \textit{clist var} \rangle$ { $\langle \textit{item} \rangle$ } { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }
<code>\clist_if_in:(NV No Ne cn cV co ce)</code>	$\langle \textit{true code} \rangle$	<code>\clist_if_in:N</code>	$\langle \textit{clist var} \rangle$ { $\langle \textit{item} \rangle$ } { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }
<code>\clist_if_in:n</code>	$\langle \textit{comma list} \rangle$	<code>\clist_if_in:n</code>	$\langle \textit{comma list} \rangle$ { $\langle \textit{item} \rangle$ } { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }
<code>\clist_if_in:(nV no ne)</code>	$\langle \textit{true code} \rangle$	<code>\clist_if_in:n</code>	$\langle \textit{comma list} \rangle$ { $\langle \textit{item} \rangle$ } { $\langle \textit{true code} \rangle$ } { $\langle \textit{false code} \rangle$ }

Tests if the $\langle \textit{item} \rangle$ is present in the $\langle \textit{clist var} \rangle$. In the case of an n-type $\langle \textit{comma list} \rangle$, the usual rules of space trimming and brace stripping apply. Hence,

`\clist_if_in:n` { a , {b}~ , {b} , c } { b } {true} {false}

yields true.

TeXhackers note: The function may fail if the $\langle \textit{item} \rangle$ contains {, }, or # (assuming the usual TeX category codes apply).

24.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e., any local assignments made by the $\langle \textit{function} \rangle$ or $\langle \textit{code} \rangle$ discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is {a, {b}, {c}}, then the arguments passed to the mapped function are ‘a’, ‘b’, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

<code>\clist_map_function:N</code>	☆	<code>\clist_map_function:N</code>	$\langle \textit{clist var} \rangle$ $\langle \textit{function} \rangle$
<code>\clist_map_function:c</code>	☆	Applies $\langle \textit{function} \rangle$ to every $\langle \textit{item} \rangle$ stored in the $\langle \textit{clist var} \rangle$. The $\langle \textit{function} \rangle$ receives one argument for each iteration. The $\langle \textit{items} \rangle$ are returned from left to right. The function <code>\clist_map_inline:N</code> is in general more efficient than <code>\clist_map_function:N</code> .	
<code>\clist_map_function:n</code>	☆		
<code>\clist_map_function:e</code>	☆		

<code>\clist_map_inline:Nn</code>	<code>\clist_map_inline:Nn <clist var> {<inline function>}</code>
<code>\clist_map_inline:cn</code>	
<code>\clist_map_inline:nn</code>	Applies <i><inline function></i> to every <i><item></i> stored within the <i><clist var></i> . The <i><inline function></i> should consist of code which receives the <i><item></i> as #1. The <i><items></i> are returned from left to right.
<code>\clist_map_variable:NNn</code>	<code>\clist_map_variable:NNn <clist var> <variable> {<code>}</code>
<code>\clist_map_variable:cNn</code>	
<code>\clist_map_variable:nNn</code>	Stores each <i><item></i> of the <i><clist var></i> in turn in the (token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. Its value after the loop is the last <i><item></i> in the <i><clist var></i> , or its original value if there were no <i><item></i> . The <i><items></i> are returned from left to right.
<code>\clist_map_tokens:Nn</code>	☆ <code>\clist_map_tokens:Nn <clist var> {<code>}</code>
<code>\clist_map_tokens:cn</code>	☆ <code>\clist_map_tokens:nn {<comma list>} {<code>}</code>
<code>\clist_map_tokens:nn</code>	☆ Calls <i><code></i> { <i><item></i> } for every <i><item></i> stored in the <i><clist var></i> . The <i><code></i> receives each <i><item></i> as a trailing brace group. If the <i><code></i> consists of a single function this is equivalent to <code>\clist_map_function:nN</code> .
	New: 2021-05-05
<code>\clist_map_break:</code>	☆ <code>\clist_map_break:</code>
	Used to terminate a <code>\clist_map...</code> function before all entries in the <i><comma list></i> have been processed. This normally takes place within a conditional statement, for example
	<pre> \clist_map_inline:Nn \l_my_clist { \str_if_eq:nnTF { #1 } { bingo } { \clist_map_break: } { % Do something useful } } </pre>
	Use outside of a <code>\clist_map...</code> scenario leads to low level T _E X errors.
	T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆ `\clist_map_break:n {<code>}`

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ★ `\clist_count:N <clist var>`

`\clist_count:c` ★ Leaves the number of items in the *<clist var>* in the input stream as an *<integer denotation>*. The total number of items in a *<clist var>* includes those which are
`\clist_count:n` ★
`\clist_count:e` ★ duplicates, i.e., every item in a *<clist var>* is counted.

24.6 Using the content of comma lists directly

`\clist_use:Nnnn` ★ `\clist_use:Nnnn <clist var> {<separator between two>}`

`\clist_use:cnnn` ★ `{<separator between more than two>} {<separator between final two>}`

Places the contents of the *<clist var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the comma list has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the comma list has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an e-type or x-type argument expansion.

<code>\clist_use:Nn</code>	★	<code>\clist_use:Nn <clist var> {<separator>}</code>
<code>\clist_use:cn</code>	★	Places the contents of the <code><clist var></code> in the input stream, with the <code><separator></code> between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\clist_use:N</code>	★	<code>\clist_use:N <clist var></code>
<code>\clist_use:c</code>	★	Places the contents of the <code><clist var></code> in the input stream, with a comma between each item. The result is exactly the stored <code><clist></code> , which will include braces around (for example) entries with retained spaces at the ends.

New: 2024-11-12

TeXhackers note: The result is returned as-is, in the same way as `\tl_use:N` and *without* protection from expansion, cf. `\clist_use:Nnnnn`, etc. It is equivalent to `V`-type expansion of a `clist`.

<code>\clist_use:nnnn</code>	★	<code>\clist_use:nnnn {<comma list>} {<separator between two>}</code>
<code>\clist_use:nn</code>	★	<code>{<separator between more than two>} {<separator between final two>}</code>
		<code>\clist_use:nn {<comma list>} {<separator>}</code>

New: 2021-05-10

Places the contents of the `<comma list>` in the input stream, with the appropriate `<separator>` between the items. As for `\clist_set:Nn`, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The `<separators>` are then inserted in the same way as for `\clist_use:Nnnn` and `\clist_use:Nn`, respectively.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an `e`-type or `x`-type argument expansion.

24.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

<code>\clist_get:NN</code> <code>\clist_get:cN</code> <code>\clist_get:NNTF</code> <code>\clist_get:cNTF</code>	<code>\clist_get:NN</code> $\langle \textit{clist var} \rangle$ $\langle \textit{tl var} \rangle$ Stores the left-most item from a $\langle \textit{clist var} \rangle$ in the $\langle \textit{tl var} \rangle$ without removing it from the $\langle \textit{clist var} \rangle$. The $\langle \textit{tl var} \rangle$ is assigned locally. In the non-branching version, if the $\langle \textit{clist var} \rangle$ is empty the $\langle \textit{tl var} \rangle$ is set to the marker value <code>\q_no_value</code> .

<code>\clist_pop:NN</code> <code>\clist_pop:cN</code>	<code>\clist_pop:NN</code> $\langle \textit{clist var} \rangle$ $\langle \textit{tl var} \rangle$ Pops the left-most item from a $\langle \textit{clist var} \rangle$ into the $\langle \textit{tl var} \rangle$, i.e., removes the item from the comma list and stores it in the $\langle \textit{tl var} \rangle$. Both of the variables are assigned locally.

<code>\clist_gpop:NN</code> <code>\clist_gpop:cN</code>	<code>\clist_gpop:NN</code> $\langle \textit{clist var} \rangle$ $\langle \textit{tl var} \rangle$ Pops the left-most item from a $\langle \textit{clist var} \rangle$ into the $\langle \textit{tl var} \rangle$, i.e., removes the item from the comma list and stores it in the $\langle \textit{tl var} \rangle$. The $\langle \textit{clist var} \rangle$ is modified globally, while the assignment of the $\langle \textit{tl var} \rangle$ is local.

<code>\clist_pop:NNTF</code> <code>\clist_pop:cNTF</code>	<code>\clist_pop:NNTF</code> $\langle \textit{clist var} \rangle$ $\langle \textit{tl var} \rangle$ $\{\langle \textit{true code} \rangle\}$ $\{\langle \textit{false code} \rangle\}$ If the $\langle \textit{clist var} \rangle$ is empty, leaves the $\langle \textit{false code} \rangle$ in the input stream. The value of the $\langle \textit{tl var} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \textit{clist var} \rangle$ is non-empty, pops the top item from the $\langle \textit{clist var} \rangle$ in the $\langle \textit{tl var} \rangle$, i.e., removes the item from the $\langle \textit{clist var} \rangle$. Both the $\langle \textit{clist var} \rangle$ and the $\langle \textit{tl var} \rangle$ are assigned locally.

<code>\clist_gpop:NNTF</code> <code>\clist_gpop:cNTF</code>	<code>\clist_gpop:NNTF</code> $\langle \textit{clist var} \rangle$ $\langle \textit{tl var} \rangle$ $\{\langle \textit{true code} \rangle\}$ $\{\langle \textit{false code} \rangle\}$ If the $\langle \textit{clist var} \rangle$ is empty, leaves the $\langle \textit{false code} \rangle$ in the input stream. The value of the $\langle \textit{tl var} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \textit{clist var} \rangle$ is non-empty, pops the top item from the $\langle \textit{clist var} \rangle$ in the $\langle \textit{tl var} \rangle$, i.e., removes the item from the $\langle \textit{clist var} \rangle$. The $\langle \textit{clist var} \rangle$ is modified globally, while the $\langle \textit{tl var} \rangle$ is assigned locally.

<code>\clist_push:Nn</code> <code>\clist_push:(NV No cn cV co)</code> <code>\clist_gpush:Nn</code> <code>\clist_gpush:(NV No cn cV co)</code>	<code>\clist_push:Nn</code> $\langle \textit{clist var} \rangle$ $\{\langle \textit{items} \rangle\}$

Adds the $\{\langle \textit{items} \rangle\}$ to the top of the $\langle \textit{clist var} \rangle$. Spaces are removed from both sides of each item as for any n-type comma list.

24.8 Using a single item

<code>\clist_item:Nn</code>	★ <code>\clist_item:Nn <clist var> {<int expr>}</code>
<code>\clist_item:cn</code>	★
<code>\clist_item:nn</code>	★ Indexing items in the <code><clist var></code> from 1 at the top (left), this function evaluates the
<code>\clist_item:(Vn vn on en)</code>	★ <code><int expr></code> and leaves the appropriate item from the comma list in the input stream.
	★ If the <code><int expr></code> is negative, indexing occurs from the bottom (right) of the comma
	list. When the <code><int expr></code> is larger than the number of items in the <code><clist var></code> (as
	calculated by <code>\clist_count:N</code>) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\clist_rand_item:N</code>	★ <code>\clist_rand_item:N <clist var></code>
<code>\clist_rand_item:c</code>	★ <code>\clist_rand_item:n {<comma list>}</code>
<code>\clist_rand_item:n</code>	★ Selects a pseudo-random item of the <code><clist var>/<comma list></code> . If the <code><comma list></code>
	has no item, the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `e`-type or `x`-type argument expansion.

24.9 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <clist var></code>
<code>\clist_show:c</code>	Displays the entries in the <code><clist var></code> in the terminal.
Updated: 2021-04-29	

<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
	Displays the entries in the comma list in the terminal.

<code>\clist_log:N</code>	<code>\clist_log:N <clist var></code>
<code>\clist_log:c</code>	Writes the entries in the <code><clist var></code> in the log file. See also <code>\clist_show:N</code> which
Updated: 2021-04-29	displays the result in the terminal.

<code>\clist_log:n</code>	<code>\clist_log:n {<tokens>}</code>
	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays
	the result in the terminal.

24.10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.

<code>\l_tmpa_clist</code> <code>\l_tmpb_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_clist</code> <code>\g_tmpb_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 25

The l3token module

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T_EX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, T_EX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }  
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section [25.7](#).

25.1 Creating character tokens

	<code>\char_set_active_eq:NN</code>	<code>\char_set_active_eq:NN</code>	$\langle char \rangle$	$\langle function \rangle$
	<code>\char_set_active_eq:Nc</code>			
	<code>\char_gset_active_eq:NN</code>			
	<code>\char_gset_active_eq:Nc</code>			

Sets the behavior of the $\langle char \rangle$ in situations where it is active (category code 13) to be equivalent to that of the definition of the $\langle function \rangle$ at the time `\char_set_active_eq:NN` is used. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

	<code>\char_set_active_eq:nN</code>	<code>\char_set_active_eq:nN</code>	$\{\langle integer\ expression \rangle\}$	$\langle function \rangle$
	<code>\char_set_active_eq:nc</code>			
	<code>\char_gset_active_eq:nN</code>			
	<code>\char_gset_active_eq:nc</code>			

Sets the behavior of the $\langle char \rangle$ which has character code as given by the $\langle integer\ expression \rangle$ in situations where it is active (category code 13) to be equivalent to that of the $\langle function \rangle$ at the time `\char_set_active_eq:nN` is used. The category code of the $\langle char \rangle$ is *unchanged* by this process. The $\langle function \rangle$ may itself be an active character.

	<code>\char_generate:nn</code>	$\star$	<code>\char_generate:nn</code>	$\{\langle charcode \rangle\}$	$\{\langle catcode \rangle\}$

Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The $\langle charcode \rangle$ may be any one valid for the engine in use, except that for $\langle catcode \rangle$ 10, $\langle charcode \rangle$ 0 is not allowed. Active characters cannot be generated in older versions of $\text{\TeX}$. Another way to build token lists with unusual category codes is `\regex_replace_all:nnN` $\{.*\}$ $\{\langle replacement \rangle\}$ $\langle t1\ var \rangle$.

$\text{\TeX}$ hackers note: Exactly two expansions are needed to produce the character.

	<code>\c_catcode_active_space_tl</code>	

Token list containing one character with category code 13, (“active”), and character code 32 (space).

<code>\c_catcode_other_space_tl</code>	Token list containing one character with category code 12, (“other”), and character code 32 (space).

25.2 Manipulating and interrogating character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n</code> $\{ \langle integer expression \rangle \}$
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

	<code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>	These functions set the category code of the <i>⟨character⟩</i> which has character code as given by the <i>⟨integer expression⟩</i> . The first <i>⟨integer expression⟩</i> is the character code and the second is the category code to apply. The setting applies within the current T _E X group. In general, the symbolic functions <code>\char_set_catcode_⟨type⟩</code> should be preferred, but there are cases where these lower-level functions may be useful.
	<code>\char_value_catcode:n</code> ★	<code>\char_value_catcode:n {⟨integer expression⟩}</code>	Expands to the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
	<code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>	Displays the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
	<code>\char_set_lccode:nn</code>	<code>\char_set_lccode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>	Sets up the behavior of the <i>⟨character⟩</i> when found inside <code>\text_lowercase:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code: <pre> \char_set_lccode:nn { ‘\A } { ‘\a } % Standard behavior \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre> The setting applies within the current T _E X group.
	<code>\char_value_lccode:n</code> ★	<code>\char_value_lccode:n {⟨integer expression⟩}</code>	Expands to the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
	<code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>	Displays the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
	<code>\char_set_uccode:nn</code>	<code>\char_set_uccode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>	Sets up the behavior of the <i>⟨character⟩</i> when found inside <code>\text_uppercase:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code: <pre> \char_set_uccode:nn { ‘\a } { ‘\A } % Standard behavior \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre> The setting applies within the current T _E X group.

<code>\char_value_uccode:n</code> *	<code>\char_value_uccode:n {⟨integer expression⟩}</code>
	Expands to the current upper case code of the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> .
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {⟨integer expression⟩}</code>
	Displays the current upper case code of the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> on the terminal.
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>
	This function sets up the math code of <code>⟨character⟩</code> . The <code>⟨character⟩</code> is specified as an <code>⟨integer expression⟩</code> which will be used as the character code of the relevant character. The setting applies within the current $\text{\TeX}$ group.
<code>\char_value_mathcode:n</code> *	<code>\char_value_mathcode:n {⟨integer expression⟩}</code>
	Expands to the current math code of the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> .
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {⟨integer expression⟩}</code>
	Displays the current math code of the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> on the terminal.
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}</code>
	This function sets up the space factor for the <code>⟨character⟩</code> . The <code>⟨character⟩</code> is specified as an <code>⟨integer expression⟩</code> which will be used as the character code of the relevant character. The setting applies within the current $\text{\TeX}$ group.
<code>\char_value_sfcode:n</code> *	<code>\char_value_sfcode:n {⟨integer expression⟩}</code>
	Expands to the current space factor for the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> .
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {⟨integer expression⟩}</code>
	Displays the current space factor for the <code>⟨character⟩</code> with character code given by the <code>⟨integer expression⟩</code> on the terminal.
<code>\l_char_active_seq</code>	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category <code>⟨active⟩</code> (catcode 13). Each entry in the sequence consists of a single escaped token, for example <code>\~</code> . Active tokens should be added to the sequence when they are defined for general document use.
<code>\l_char_special_seq</code>	Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories <code>⟨letter⟩</code> (catcode 11) or <code>⟨other⟩</code> (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

25.3 Generic tokens

<code>\c_group_begin_token</code>	These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.
<code>\c_group_end_token</code>	
<code>\c_math_toggle_token</code>	
<code>\c_alignment_token</code>	
<code>\c_parameter_token</code>	
<code>\c_math_superscript_token</code>	
<code>\c_math_subscript_token</code>	T_EXhackers note: The tokens <code>\c_group_begin_token</code> , <code>\c_group_end_token</code> , and <code>\c_space_token</code> are expl3 counterparts of L ^A T _E X 2 _ε 's <code>\bgroup</code> , <code>\egroup</code> , and <code>\@sptoken</code> .
<code>\c_space_token</code>	

<code>\c_catcode_letter_token</code>	These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.
<code>\c_catcode_other_token</code>	

25.4 Converting tokens

<code>\token_to_meaning:N</code> *	<code>\token_to_meaning:N</code> $\langle token \rangle$
<code>\token_to_meaning:c</code> *	Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This is the primitive T _E X description of the $\langle token \rangle$, thus for example both functions defined by <code>\cs_set_nopar:Npn</code> and token list variables defined using <code>\tl_new:N</code> are described as macros.

T_EXhackers note: This is the T_EX primitive `\meaning`. The $\langle token \rangle$ can thus be an explicit space token or an explicit begin-group or end-group character token (`{` or `}` when normal T_EX category codes apply) even though these are not valid N-type arguments.

<code>\token_to_str:N</code> *	<code>\token_to_str:N</code> $\langle token \rangle$
<code>\token_to_str:c</code> *	Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). If the $\langle token \rangle$ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

T_EXhackers note: `\token_to_str:N` is the T_EX primitive `\string`. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal T_EX category codes apply) even though these are not valid N-type arguments.

<code>\token_to_catcode:N</code> *	<code>\token_to_catcode:N</code> $\langle token \rangle$
New: 2023-10-15	Converts the given $\langle token \rangle$ into a number describing its category code. If $\langle token \rangle$ is a control sequence this expands to 16. This can't detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

25.5 Token conditionals

<code>\token_if_group_begin_p:N</code>	<code>*</code>	<code>\token_if_group_begin_p:N</code>	<code><token></code>
<code>\token_if_group_begin:NTF</code>	<code>*</code>	<code>\token_if_group_begin:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a begin group token (`{` when normal `TEX` category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_group_end_p:N</code>	<code>*</code>	<code>\token_if_group_end_p:N</code>	<code><token></code>
<code>\token_if_group_end:NTF</code>	<code>*</code>	<code>\token_if_group_end:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an end group token (`}` when normal `TEX` category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_math_toggle_p:N</code>	<code>*</code>	<code>\token_if_math_toggle_p:N</code>	<code><token></code>
<code>\token_if_math_toggle:NTF</code>	<code>*</code>	<code>\token_if_math_toggle:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a math shift token (`$` when normal `TEX` category codes are in force).

<code>\token_if_alignment_p:N</code>	<code>*</code>	<code>\token_if_alignment_p:N</code>	<code><token></code>
<code>\token_if_alignment:NTF</code>	<code>*</code>	<code>\token_if_alignment:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an alignment token (`&` when normal `TEX` category codes are in force).

<code>\token_if_parameter_p:N</code>	<code>*</code>	<code>\token_if_parameter_p:N</code>	<code><token></code>
<code>\token_if_parameter:NTF</code>	<code>*</code>	<code>\token_if_parameter:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a macro parameter token (`#` when normal `TEX` category codes are in force).

<code>\token_if_math_superscript_p:N</code>	<code>*</code>	<code>\token_if_math_superscript_p:N</code>	<code><token></code>
<code>\token_if_math_superscript:NTF</code>	<code>*</code>	<code>\token_if_math_superscript:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a superscript token (`^` when normal `TEX` category codes are in force).

<code>\token_if_math_subscript_p:N</code>	<code>*</code>	<code>\token_if_math_subscript_p:N</code>	<code><token></code>
<code>\token_if_math_subscript:NTF</code>	<code>*</code>	<code>\token_if_math_subscript:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a subscript token (`_` when normal `TEX` category codes are in force).

<code>\token_if_space_p:N</code>	<code>*</code>	<code>\token_if_space_p:N</code>	<code><token></code>
<code>\token_if_space:NTF</code>	<code>*</code>	<code>\token_if_space:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

```
\token_if_letter_p:N * \token_if_letter_p:N <token>
\token_if_letter:NTF * \token_if_letter:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a letter token.

```
\token_if_other_p:N * \token_if_other_p:N <token>
\token_if_other:NTF * \token_if_other:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an “other” token.

```
\token_if_active_p:N * \token_if_active_p:N <token>
\token_if_active:NTF * \token_if_active:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an active character.

```
\token_if_eq_catcode_p:NN * \token_if_eq_catcode_p:NN <token1> <token2>
\token_if_eq_catcode:NNTF * \token_if_eq_catcode:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two $\langle tokens \rangle$ have the same category code.

```
\token_if_eq_charcode_p:NN * \token_if_eq_charcode_p:NN <token1> <token2>
\token_if_eq_charcode:NNTF * \token_if_eq_charcode:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two $\langle tokens \rangle$ have the same character code.

```
\token_if_eq_meaning_p:NN * \token_if_eq_meaning_p:NN <token1> <token2>
\token_if_eq_meaning:NNTF * \token_if_eq_meaning:NNTF <token1> <token2> {\true code} {\false code}
```

Tests if the two $\langle tokens \rangle$ have the same meaning when expanded.

```
\token_if_macro_p:N * \token_if_macro_p:N <token>
\token_if_macro:NTF * \token_if_macro:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is a $\text{\TeX}$ macro.

```
\token_if_cs_p:N * \token_if_cs_p:N <token>
\token_if_cs:NTF * \token_if_cs:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is a control sequence.

```
\token_if_expandable_p:N * \token_if_expandable_p:N <token>
\token_if_expandable:NTF * \token_if_expandable:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is expandable. This test returns $\langle false \rangle$ for an undefined token.

```
\token_if_control_symbol_p:N * \token_if_control_symbol_p:N <token>
\token_if_control_symbol:NTF * \token_if_control_symbol:NTF <token> {\true code} {\false code}
```

New: 2025-05-12

Tests whether the $\langle token \rangle$ is a control sequence with a name comprised of exactly one non-letter character (called a “control symbol”). Specifically, only the following tokens leave $\langle false code \rangle$:

- explicit characters, such as `a` or `"`
- the escape character followed by one or more characters of category code 11 (letter), such as `\foo`

Any other token will leave $\langle true code \rangle$. The category codes which apply are those at the point the test is used, not those used when the $\langle token \rangle$ is defined.

```
\token_if_control_word_p:N * \token_if_control_word_p:N <token>
\token_if_control_word:NTF * \token_if_control_word:NTF <token> {\true code} {\false code}
```

New: 2025-05-12

Tests whether the $\langle token \rangle$ is a control sequence with a name comprised of one or more letters (called a “control word”). Specifically, only these tokens leave $\langle false code \rangle$:

- explicit characters, such as `a` or `"`
- the escape character followed by exactly one character whose category code is not 11 (letter) when used (not tokenized), such as `\ , \&`

Any other token will leave $\langle true code \rangle$. The category codes which apply are those at the point the test is used, not those used when the $\langle token \rangle$ is defined.

```
\token_if_long_macro_p:N * \token_if_long_macro_p:N <token>
\token_if_long_macro:NTF * \token_if_long_macro:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is a long macro with no other prefix; to test for a macro that is both long and protected, use `\token_if_protected_long_macro:N(TF)`.

```
\token_if_protected_macro_p:N * \token_if_protected_macro_p:N <token>
\token_if_protected_macro:NTF * \token_if_protected_macro:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is a protected macro with no other prefix; to test for a macro that is both protected and long, use `\token_if_protected_long_macro:N(TF)`.

```
\token_if_protected_long_macro_p:N * \token_if_protected_long_macro_p:N <token>
\token_if_protected_long_macro:NTF * \token_if_protected_long_macro:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is a protected long macro.

```
\token_if_chardef_p:N * \token_if_chardef_p:N <token>
\token_if_chardef:NTF * \token_if_chardef:NTF <token> {\true code} {\false code}
```

Tests if the $\langle token \rangle$ is defined to be a chardef.

TeXhackers note: Booleans, boxes and small integer constants are implemented as `\chardefs`.

```

\token_if_mathchardef_p:N * \token_if_mathchardef_p:N <token>
\token_if_mathchardef:NTF * \token_if_mathchardef:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

```

\token_if_font_selection_p:N * \token_if_font_selection_p:N <token>
\token_if_font_selection:NTF * \token_if_font_selection:NTF <token> {\true code} {\false code}

```

New: 2020-10-27

Tests if the $\langle token \rangle$ is defined to be a font selection command.

```

\token_if_dim_register_p:N * \token_if_dim_register_p:N <token>
\token_if_dim_register:NTF * \token_if_dim_register:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a dimension register.

```

\token_if_int_register_p:N * \token_if_int_register_p:N <token>
\token_if_int_register:NTF * \token_if_int_register:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, $\backslash chardefs$, or $\backslash mathchardefs$ depending on their value.

```

\token_if_muskip_register_p:N * \token_if_muskip_register_p:N <token>
\token_if_muskip_register:NTF * \token_if_muskip_register:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a muskip register.

```

\token_if_skip_register_p:N * \token_if_skip_register_p:N <token>
\token_if_skip_register:NTF * \token_if_skip_register:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a skip register.

```

\token_if_toks_register_p:N * \token_if_toks_register_p:N <token>
\token_if_toks_register:NTF * \token_if_toks_register:NTF <token> {\true code} {\false code}

```

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

```

\token_if_primitive_p:N * \token_if_primitive_p:N <token>
\token_if_primitive:NTF * \token_if_primitive:NTF <token> {\true code} {\false code}

```

Updated: 2020-09-11

Tests if the $\langle token \rangle$ is an engine primitive. In LuaT_EX this includes primitive-like commands defined using `token.set_lua`.

<code>\token_case_catcode:Nn</code>	<code>*</code>	<code>\token_case_meaning:NnTF</code>	<code><test token></code>
<code>\token_case_catcode:NnTF</code>	<code>*</code>	<code>{</code>	
<code>\token_case_charcode:Nn</code>	<code>*</code>	<code><token case₁> {<code case₁>}</code>	
<code>\token_case_charcode:NnTF</code>	<code>*</code>	<code><token case₂> {<code case₂>}</code>	
<code>\token_case_meaning:Nn</code>	<code>*</code>	<code>...</code>	
<code>\token_case_meaning:NnTF</code>	<code>*</code>	<code><token case_n> {<code case_n>}</code>	
		<code>}</code>	
		<code>{<true code>}</code>	
		<code>{<false code>}</code>	

New: 2020-12-03

This function compares the `<test token>` in turn with each of the `<token case>`s. If the two are equal (as described for `\token_if_eq_catcode:NNTF`, `\token_if_eq_charcode:NNTF` and `\token_if_eq_meaning:NNTF`, respectively) then the associated `<code>` is left in the input stream and other cases are discarded. If any of the cases are matched, the `<true code>` is also inserted into the input stream (after the code for the appropriate case), while if none match then the `<false code>` is inserted. The functions `\token_case_catcode:Nn`, `\token_case_charcode:Nn`, and `\token_case_meaning:Nn`, which do nothing if there is no match, are also available.

25.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. Peeking ahead does *not* skip spaces: rather, `\peek_remove_spaces:n` should be used. In addition, using `\peek_analysis_map_inline:n`, one can map through the following tokens in the input stream and repeatedly perform some tests.

<code>\peek_after:Nw</code>	<code>\peek_after:Nw</code>	<code><function></code>	<code><token></code>

Locally sets the test variable `\l_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal T_EX category codes), i.e., it is not necessarily the next argument which would be grabbed by a normal function.

<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw</code>	<code><function></code>	<code><token></code>

Globally sets the test variable `\g_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal T_EX category codes), i.e., it is not necessarily the next argument which would be grabbed by a normal function.

<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.

<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.

<u>\peek_catcode:NTF</u>	\peek_catcode:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same category code as the <test token> (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the <token> is left in the input stream after the <true code> or <false code> (as appropriate to the result of the test).
<u>\peek_catcode_remove:NTF</u>	\peek_catcode_remove:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same category code as the <test token> (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).
<u>\peek_charcode:NTF</u>	\peek_charcode:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same character code as the <test token> (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the <token> is left in the input stream after the <true code> or <false code> (as appropriate to the result of the test).
<u>\peek_charcode_remove:NTF</u>	\peek_charcode_remove:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same character code as the <test token> (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).
<u>\peek_meaning:NTF</u>	\peek_meaning:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <token> is left in the input stream after the <true code> or <false code> (as appropriate to the result of the test).
<u>\peek_meaning_remove:NTF</u>	\peek_meaning_remove:NTF <test token> {<true code>} {<false code>}	Tests if the next <token> in the input stream has the same meaning as the <test token> (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the <token> is removed from the input stream if the test is true. The function then places either the <true code> or <false code> in the input stream (as appropriate to the result of the test).
<u>\peek_remove_spaces:n</u>	\peek_remove_spaces:n {<code>}	Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the <code> will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.

<code>\peek_remove_filler:n</code>	<code>\peek_remove_filler:n {<code>}</code>

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using f-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the `<code>` will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

T_EXhackers note: This is essentially a macro-based implementation of how T_EX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the `<filler>` (i.e., it does not lead to a T_EX error).

In contrast to T_EX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

<code>\peek_N_type:TF</code>	<code>\peek_N_type:TF {<true code>} {<false code>}</code>

Tests if the next `<token>` in the input stream can be safely grabbed as an N-type argument. The test is `<false>` if the next `<token>` is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and `<true>` in all other cases. Note that a `<true>` result ensures that the next `<token>` is a valid N-type argument. However, if the next `<token>` is for instance `\c_space_token`, the test takes the `<false>` branch, even though the next `<token>` is in fact a valid N-type argument. The `<token>` is left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

<code>\peek_analysis_map_inline:n</code>	<code>\peek_analysis_map_inline:n {(inline function)}</code>

New: 2020-12-03

Updated: 2024-02-07

Repeatedly removes one $\langle token \rangle$ from the input stream and applies the $\langle inline function \rangle$ to it, until `\peek_analysis_map_break:` is called. The $\langle inline function \rangle$ receives three arguments for each $\langle token \rangle$ in the input stream:

- $\langle tokens \rangle$, which both `o`-expand and `e/x`-expand to the $\langle token \rangle$. The detailed form of $\langle tokens \rangle$ may change in later releases.
- $\langle char code \rangle$, a decimal representation of the character code of the $\langle token \rangle$, -1 if it is a control sequence.
- $\langle catcode \rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token \rangle$ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing " $\langle catcode \rangle$ ".

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in `l3tl-analysis`. The $\langle char code \rangle$ and $\langle catcode \rangle$ do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the $\langle inline function \rangle$ with #1 being `\exp_not:n { \c_group_begin_token }` (with the current implementation), #2 being -1 , and #3 being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token `{`, the $\langle inline function \rangle$ is called with arguments `\exp_after:wN { \if_false: } \fi:`, 123 and 1.

The mapping is done at the current group level, i.e., any local assignments made by the $\langle inline function \rangle$ remain in effect after the loop. Within the code, `\l_peek_token` is set equal (as a token, not a token list) to the token under consideration.

Peek functions cannot be used within this mapping function (nor other mapping functions) since the input stream contains trailing material necessary for the functioning of the loop.

T_EXhackers note: In case the input stream has not yet been tokenized (converted from characters to tokens), characters are tokenized one by one as needed by `\peek_analysis_map_inline:n` using the current category code régime.

<code>\peek_analysis_map_break:</code>	<code>\peek_analysis_map_inline:n</code>
<code>\peek_analysis_map_break:n</code>	<code>{ ... \peek_analysis_map_break:n {(code)} }</code>

New: 2020-12-03

Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts $\langle code \rangle$ in the input stream (empty for `\peek_analysis_map_break:`).

<u>\peek_regex:nTF</u>	\peek_regex:nTF {<regex>} {<true code>} {<false code>}
<u>\peek_regex:NTF</u>	Tests if the <tokens> that follow in the input stream match the <regular expression>. Any <tokens> that have been read are left in the input stream after the <true code> or <false code> (as appropriate to the result of the test). See l3regex for documentation of the syntax of regular expressions. The <regular expression> is implicitly anchored at the start, so for instance \peek_regex:nTF { a } is essentially equivalent to \peek_charcode:NTF a.
New: 2020-12-03	

T_EXhackers note: Implicit character tokens are correctly considered by \peek_regex:nTF as control sequences, while functions that inspect individual tokens (for instance \peek_charcode:NTF) only take into account their meaning.

The \peek_regex:nTF function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance \peek_regex:nTF { abc | [a-z] } { } { } abc will only inspect the first token a even though the first branch abc of the alternative is preferred in functions such as \peek_regex_remove_once:nTF. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

<u>\peek_regex_remove_once:nTF</u>	\peek_regex_remove_once:nTF {<regex>} {<true code>} {<false code>}
<u>\peek_regex_remove_once:NTF</u>	

New: 2020-12-03

Tests if the <tokens> that follow in the input stream match the <regex>. If the test is true, the <tokens> are removed from the input stream and the <true code> is inserted, while if the test is false, the <false code> is inserted followed by the <tokens> that were originally in the input stream. See l3regex for documentation of the syntax of regular expressions. The <regular expression> is implicitly anchored at the start, so for instance \peek_regex_remove_once:nTF { a } is essentially equivalent to \peek_charcode_remove:NTF a.

T_EXhackers note: Implicit character tokens are correctly considered by \peek_regex_remove_once:nTF as control sequences, while functions that inspect individual tokens (for instance \peek_charcode:NTF) only take into account their meaning.

```

\peek_regex_replace_once:nn    \peek_regex_replace_once:nnTF {\<regex>} {\<replacement>} {\<true code>} {\<false
\peek_regex_replace_once:nnTF code>}
\peek_regex_replace_once:Nn
\peek_regex_replace_once:NnTF

```

New: 2020-12-03

If the $\langle tokens \rangle$ that follow in the input stream match the $\langle regex \rangle$, replaces them according to the $\langle replacement \rangle$ as for `\regex_replace_once:nnN`, and leaves the result in the input stream, after the $\langle true code \rangle$. Otherwise, leaves $\langle false code \rangle$ followed by the $\langle tokens \rangle$ that were originally in the input stream, with no modifications. See `l3regex` for documentation of the syntax of regular expressions and of the $\langle replacement \rangle$: for instance `\0` in the $\langle replacement \rangle$ is replaced by the tokens that were matched in the input stream. The $\langle regular expression \rangle$ is implicitly anchored at the start. In contrast to `\regex_replace_once:nnN`, no error arises if the $\langle replacement \rangle$ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

TeXhackers note: Implicit character tokens are correctly considered by `\peek_regex_replace_once:nnTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:N`) only take into account their meaning.

25.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand\<token>` (when the $\langle token \rangle$ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: \<token>`, whose shape coincides with the $\langle token \rangle$ and whose meaning differs from `\relax`.

- An `\outer endtemplate`: can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of “bytes” `~~~~~1100xy` where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from `"11 0000 = 1 114 112` to `"110 0ff = 1 114 367`. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the_ character_` followed by the given byte. If this byte is in the range `80–ff` this gives an “invalid utf-8 sequence” error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L^AT_EX3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),
- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

Category code 13 (*active*) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in L^AT_EX3 for most functions and some variables (`t1`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in L^AT_EX3 for some functions,
- a register such as `\count123`, used in L^AT_EX3 for the implementation of some variables (`int`, `dim`, ...),

- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what L^AT_EX3 calls `nopar`), and `\outer` or not (unused in L^AT_EX3). Their `\meaning` takes the form

`⟨prefix⟩ macro:⟨argument⟩->⟨replacement⟩`

where `⟨prefix⟩` is among `\protected\long\outer`, `⟨argument⟩` describes parameters that the macro expects, such as `#1#2#3`, and `⟨replacement⟩` describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument T_EX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Chapter 26

The l3prop module

Property lists

expl3 implements a “property list” data type, which contains an unordered list of entries each of which consists of a $\langle key \rangle$ (string) and an associated $\langle value \rangle$ (token list). The $\langle key \rangle$ and $\langle value \rangle$ may both be given as any balanced text, and the $\langle key \rangle$ is processed using `\tl_to_str:n`, meaning that category codes are ignored. Entries can be manipulated individually, as well as collectively by applying a function to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nnTF`.

Property lists are intended for storing key-based information for use within code. They can be converted from and to key–value lists, which are a form of *input* parsed by the `l3keys` module. If a key–value list contains a $\langle key \rangle$ multiple times, only the last $\langle value \rangle$ associated to it will be kept in the conversion to a property list.

Internally, property lists can use two distinct implementations with different data storage, which are decided when declaring the property list variable using `\prop_new:N` (“flat” storage) or `\prop_new_linked:N` (“linked” storage). After a property list is declared with `\prop_new:N` or `\prop_new_linked:N`, the type of internal data storage can be changed by `\prop_make_flat:N` or `\prop_make_linked:N`, but only at the outermost group level. All other `l3prop` functions transparently manipulate either storage method and convert as needed.

- The (default) “flat” storage method is suited for a relatively small number of entries, or when the property list is likely to be manipulated (copied, mapped) as a whole rather than entry-wise. It is significantly faster for `\prop_set_eq:NN`, and only slightly faster for `\prop_clear:N`, `\prop_concat:NNN`, and mapping functions `\prop_map_....`
- The “linked” storage method is meant for property lists with a large numbers of entries. It takes up more of TeX’s memory during a run, but is significantly faster (for long lists) when accessing or modifying individual entries using functions such as `\prop_if_in:Nn`, `\prop_item:Nn`, `\prop_put:Nnn`, `\prop_get:NnN`, `\prop_pop:NnN`, `\prop_remove:Nn`, as it takes a constant time for these operations (rather

than the number of items for a “flat” property list). A technical drawback is that memory is permanently used⁷ by $\langle\text{keys}\rangle$ stored in a “linked” property list, even after they are removed and the property list is deleted.

26.1 Creating and initializing property lists

$\backslash\text{prop\_new:N}$	$\backslash\text{prop\_new:N } \langle\text{property list}\rangle$
$\backslash\text{prop\_new:c}$	Creates a new “flat” $\langle\text{property list}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\text{property list}\rangle$ initially contains no entries. See also $\backslash\text{prop\_new\_linked:N}$.

$\backslash\text{prop\_new\_linked:N}$	$\backslash\text{prop\_new\_linked:N } \langle\text{property list}\rangle$
$\backslash\text{prop\_new\_linked:c}$	Creates a new “linked” $\langle\text{property list}\rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle\text{property list}\rangle$ initially contains no entries. The internal data storage differs from that produced by $\backslash\text{prop\_new:N}$ and it is optimized for property lists with a large number of entries.

New: 2024-02-12

$\backslash\text{prop\_clear:N}$	$\backslash\text{prop\_clear:N } \langle\text{property list}\rangle$
$\backslash\text{prop\_clear:c}$	
$\backslash\text{prop\_gclear:N}$	Clears all entries from the $\langle\text{property list}\rangle$.
$\backslash\text{prop\_gclear:c}$	

$\backslash\text{prop\_clear\_new:N}$	$\backslash\text{prop\_clear\_new:N } \langle\text{property list}\rangle$
$\backslash\text{prop\_clear\_new:c}$	
$\backslash\text{prop\_gclear\_new:N}$	Ensures that the $\langle\text{property list}\rangle$ exists globally by applying $\backslash\text{prop\_new:N}$ if necessary, then applies $\backslash\text{prop\_}(g)\text{clear:N}$ to leave the list empty.
$\backslash\text{prop\_gclear\_new:c}$	

T_EXhackers note: If the property list exists and is of “linked” type, it is cleared but not made into a flat property list.

$\backslash\text{prop\_clear\_new\_linked:N}$	$\backslash\text{prop\_clear\_new\_linked:N } \langle\text{property list}\rangle$
$\backslash\text{prop\_clear\_new\_linked:c}$	
$\backslash\text{prop\_gclear\_new\_linked:N}$	Ensures that the $\langle\text{property list}\rangle$ exists globally by applying $\backslash\text{prop\_new\_linked:N}$ if necessary, then applies $\backslash\text{prop\_}(g)\text{clear:N}$ to leave the list empty.
$\backslash\text{prop\_gclear\_new\_linked:c}$	

New: 2024-02-12

T_EXhackers note: If the property list exists and is of “flat” type, it is cleared but not made into a linked property list.

$\backslash\text{prop\_set\_eq:NN}$	$\backslash\text{prop\_set\_eq:NN } \langle\text{property list}_1\rangle \langle\text{property list}_2\rangle$
$\backslash\text{prop\_set\_eq:(cN Nc cc)}$	
$\backslash\text{prop\_gset\_eq:NN}$	Sets the content of $\langle\text{property list}_1\rangle$ equal to that of $\langle\text{property list}_2\rangle$. This converts as needed between the two storage types.
$\backslash\text{prop\_gset\_eq:(cN Nc cc)}$	

⁷Until the end of the run, that is.

<code>\prop_set_from_keyval:Nn</code> <code>\prop_set_from_keyval:cn</code> <code>\prop_gset_from_keyval:Nn</code> <code>\prop_gset_from_keyval:cn</code>	<code>\prop_set_from_keyval:Nn <property list></code> <code>{</code> <code> <key₁> = <value₁> ,</code> <code> <key₂> = <value₂> , ...</code> <code>}</code>

Updated: 2021-11-07

Sets `<property list>` to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept. In contrast to most keyval lists (e.g. those in `l3keys`), each key here *must* be followed with an = sign even to specify an empty `<value>`.

Spaces are trimmed around every `<key>` and every `<value>`, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the `<key>` and the `<value>` to contain spaces, commas or equal signs. The `<key>` is then processed by `\tl_to_str:n`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

<code>\prop_const_from_keyval:Nn</code> <code>\prop_const_from_keyval:cn</code>	<code>\prop_const_from_keyval:Nn <property list></code> <code>{</code> <code> <key₁> = <value₁> ,</code> <code> <key₂> = <value₂> , ...</code> <code>}</code>

Updated: 2021-11-07

Creates a new constant “flat” `<property list>` or raises an error if the name is already taken. The `<property list>` is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

<code>\prop_const_linked_from_keyval:Nn</code> <code>\prop_const_linked_from_keyval:cn</code>	<code>\prop_const_linked_from_keyval:Nn <property list></code> <code>{</code> <code> <key₁> = <value₁> ,</code> <code> <key₂> = <value₂> , ...</code> <code>}</code>

New: 2024-02-12

Creates a new constant “linked” `<property list>` or raises an error if the name is already taken. The `<property list>` is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

<code>\prop_make_flat:N</code> <code>\prop_make_flat:c</code>	<code>\prop_make_flat:N <property list></code>

New: 2024-02-12

Changes the internal storage type of the `<property list>` to be the same “flat” storage as `\prop_new:N`. The key–value pairs of the `<property list>` are preserved by the change. If the property list was already flat then nothing is done. This function can only be used at the outermost group level.

<code>\prop_make_linked:N</code> <code>\prop_make_linked:c</code>	<code>\prop_make_linked:N <property list></code>

New: 2024-02-12

Changes the internal storage type of the `<property list>` to be the same “linked” storage as `\prop_new_linked:N`. The key–value pairs of the `<property list>` are preserved by the change. If the property list was already linked then nothing is done. This function can only be used at the outermost group level.

26.2 Adding and updating property list entries

<code>\prop_put:Nnn</code>	<code>\prop_put:Nnn <property list> {(key)} {(value)}</code>
<code>\prop_put:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee Nno Non Noo cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee cno con coo)</code>	
<code>\prop_gput:Nnn</code>	
<code>\prop_gput:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee Nno Non Noo cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee cno con coo)</code>	

Adds an entry to the `<property list>` which may be accessed using the `<key>` and which has `<value>`. If the `<key>` is already present in the `<property list>`, the existing entry is overwritten by the new `<value>`. Both the `<key>` and `<value>` may contain any `<balanced text>`. The `<key>` is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

<code>\prop_put_if_not_in:Nnn</code>	<code>\prop_put_if_not_in:Nnn <property list> {(key)} {(value)}</code>
<code>\prop_put_if_not_in:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee)</code>	
<code>\prop_gput_if_not_in:Nnn</code>	
<code>\prop_gput_if_not_in:(NnV Nnv Nne NVn NVV NVv Nve Nvn NvV Nvv Nve Nen NeV Nev Nee cnn cnV cnv cne cVn cVV cVv cVe cvn cvV cvv cve cen ceV cev cee)</code>	

New: 2024-03-30
Updated: 2024-05-07

If the `<key>` is present in the `<property list>` then no action is taken. Otherwise, a new entry is added as described for `\prop_put:Nnn`.

<code>\prop_concat:NNN</code>	<code>\prop_concat:NNN <property list₁> <property list₂> <property list₃></code>
<code>\prop_concat:ccc</code>	
<code>\prop_gconcat:NNN</code>	
<code>\prop_gconcat:ccc</code>	Combines the key–value pairs of <code><property list₂></code> and <code><property list₃></code> , and saves the result in <code><property list₁></code> . If a key appears in both <code><property list₂></code> and <code><property list₃></code> then the last value, namely the value in <code><property list₃></code> is kept. This converts as needed between the two storage types.
New: 2021-05-16	

<code>\prop_put_from_keyval:Nn</code>	<code>\prop_put_from_keyval:Nn <property list></code>
<code>\prop_put_from_keyval:cn</code>	{
<code>\prop_gput_from_keyval:Nn</code>	<code><key₁> = <value₁> ,</code>
<code>\prop_gput_from_keyval:cn</code>	<code><key₂> = <value₂> , ...</code>
	}

New: 2021-05-16
Updated: 2021-11-07

Updates the `<property list>` by adding entries for each key–value pair given in the second argument. The addition is done through `\prop_put:Nnn`, hence if the `<property list>` already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property list using `\prop_set_from_keyval:Nn`, then combining `<property list>` with the temporary variable using `\prop_concat:NNN`. In particular, the `<keys>` and `<values>` are space-trimmed and unbraced as described in `\prop_set_from_keyval:Nn`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

26.3 Recovering values from property lists

<code>\prop_get:NnN</code>	<code>\prop_get:NnN <property list> {<key>} <t1 var></code>
<code>\prop_get:(NVN NvN NeN NoN cnN cVN cvN ceN coN cnc)</code>	

Recovers the `<value>` stored with `<key>` from the `<property list>`, and places this in the `<t1 var>`. If the `<key>` is not found in the `<property list>` then the `<t1 var>` is set to the special marker `\q_no_value`. The `<t1 var>` is set within the current $\text{\TeX}$ group. See also `\prop_get:NnNTF`.

<code>\prop_pop:NnN</code>	<code>\prop_pop:NnN <property list> {<key>} <t1 var></code>
<code>\prop_pop:(NVN NoN cnN cVN coN)</code>	

Recovers the `<value>` stored with `<key>` from the `<property list>`, and places this in the `<t1 var>`. If the `<key>` is not found in the `<property list>` then the `<t1 var>` is set to the special marker `\q_no_value`. The `<key>` and `<value>` are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

<code>\prop_gpop:NnN</code>	<code>\prop_gpop:NnN <property list> {<key>} <t1 var></code>
<code>\prop_gpop:(NVN NoN cnN cVN coN)</code>	

Recovers the `<value>` stored with `<key>` from the `<property list>`, and places this in the `<t1 var>`. If the `<key>` is not found in the `<property list>` then the `<t1 var>` is set to the special marker `\q_no_value`. The `<key>` and `<value>` are then deleted from the property list. The `<property list>` is modified globally, while the assignment of the `<t1 var>` is local. See also `\prop_gpop:NnNTF`.

<code>\prop_item:Nn</code>	<code>*</code>	<code>\prop_item:Nn <property list> {<key>}</code>
<code>\prop_item:(NV Ne No cn cV ce co)</code>	<code>*</code>	

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers note: For “flat” property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on `\prop_get:NnN`. (For “linked” property lists both functions are fast.)

The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an **e**-type or **x**-type argument expansion.

<code>\prop_count:N</code>	<code>*</code>	<code>\prop_count:N <property list></code>
<code>\prop_count:c</code>	<code>*</code>	

Leaves the number of key–value pairs in the $\langle property list \rangle$ in the input stream as an $\langle integer denotation \rangle$.

<code>\prop_to_keyval:N</code>	<code>*</code>	<code>\prop_to_keyval:N <property list></code>

Expands to the $\langle property list \rangle$ in a key–value notation. Keep in mind that a $\langle property list \rangle$ is *unordered*, while key–value interfaces are not necessarily, so this cannot be used for arbitrary interfaces.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the key–value list does not expand further when appearing in an **e**-type or **x**-type argument expansion. It also needs exactly two steps of expansion.

26.4 Modifying property lists

<code>\prop_remove:Nn</code>	<code>\prop_remove:Nn <property list> {<key>}</code>
<code>\prop_remove:(NV Ne cn cV ce)</code>	
<code>\prop_gremove:Nn</code>	
<code>\prop_gremove:(NV Ne cn cV ce)</code>	

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e.* there is no need to test for the existence of a key before deleting it.

26.5 Property list conditionals

<code>\prop_if_exist_p:N</code>	<code>*</code>	<code>\prop_if_exist_p:N <property list></code>
<code>\prop_if_exist_p:c</code>	<code>*</code>	<code>\prop_if_exist:NTF <property list> {<true code>} {<false code>}</code>
<code>\prop_if_exist:N$\overline{T}$F</code>	<code>*</code>	Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.
<code>\prop_if_exist:c$\overline{T}$F</code>	<code>*</code>	

```

\prop_if_empty_p:N * \prop_if_empty_p:N <property list>
\prop_if_empty_p:c * \prop_if_empty:NTF <property list> {\true code} {\false code}
\prop_if_empty:NTF * Tests if the <property list> is empty (containing no entries).
\prop_if_empty:cTF *

```

```

\prop_if_in_p:Nn * \prop_if_in_p:Nn <property list> {\key}
\prop_if_in_p:(NV|Ne|No|cn|cV|ce|co) * \prop_if_in:NnTF <property list> {\key} {\true code} {\false code}
\prop_if_in:NnTF *
\prop_if_in:(NV|Ne|No|cn|cV|ce|co)TF *

```

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nNTF`.

TeXhackers note: For “flat” property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on `\prop_get:NnNTF`. (For “linked” property lists both functions are fast.)

26.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different code follows depending on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

```

\prop_get:NnNTF * \prop_get:NnNTF <property list> {\key} <t1 var>
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)TF * {\true code} {\false code}

```

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle t1 var \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, stores the corresponding $\langle value \rangle$ in the $\langle t1 var \rangle$ without removing it from the $\langle property list \rangle$, then leaves the $\langle true code \rangle$ in the input stream. The $\langle t1 var \rangle$ is assigned locally.

```

\prop_pop:NnNTF * \prop_pop:NnNTF <property list> {\key} <t1 var>
\prop_pop:(NVN|NoN|cnN|cVN|coN)TF * {\true code} {\false code}

```

If the $\langle key \rangle$ is not present in the $\langle property list \rangle$, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle t1 var \rangle$ is not defined in this case and should not be relied upon. If the $\langle key \rangle$ is present in the $\langle property list \rangle$, pops the corresponding $\langle value \rangle$ in the $\langle t1 var \rangle$, i.e., removes the item from the $\langle property list \rangle$. Both the $\langle property list \rangle$ and the $\langle t1 var \rangle$ are assigned locally.

<code>\prop_gpop:NnTF</code>	<code>\prop_gpop:NnTF <property list> {<key>} <t1 var></code>
<code>\prop_gpop:(NVN NoN cnN cVN coN)TF</code>	<code>{<true code>} {<false code>}</code>

If the `<key>` is not present in the `<property list>`, leaves the `<false code>` in the input stream. The value of the `<t1 var>` is not defined in this case and should not be relied upon. If the `<key>` is present in the `<property list>`, pops the corresponding `<value>` in the `<t1 var>`, i.e., removes the item from the `<property list>`. The `<property list>` is modified globally, while the `<t1 var>` is assigned locally.

26.7 Mapping over property lists

All mappings are done at the current group level, i.e., any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

<code>\prop_map_function:Nn</code> ☆	<code>\prop_map_function:Nn <property list> <function></code>
<code>\prop_map_function:cN</code> ☆	

Applies `<function>` to every `<entry>` stored in the `<property list>`. The `<function>` receives two arguments for each iteration: the `<key>` and associated `<value>`. The order in which `<entries>` are returned is not defined and should not be relied upon. To pass further arguments to the `<function>`, see `\prop_map_inline:Nn` (non-expandable) or `\prop_map_tokens:Nn`.

<code>\prop_map_inline:Nn</code>	<code>\prop_map_inline:Nn <property list> {<inline function>}</code>
<code>\prop_map_inline:cn</code>	

Applies `<inline function>` to every `<entry>` stored within the `<property list>`. The `<inline function>` should consist of code which receives the `<key>` as #1 and the `<value>` as #2. The order in which `<entries>` are returned is not defined and should not be relied upon.

<code>\prop_map_tokens:Nn</code> ☆	<code>\prop_map_tokens:Nn <property list> {<code>}</code>
<code>\prop_map_tokens:cn</code> ☆	

Analogue of `\prop_map_function:Nn` which maps several tokens instead of a single function. The `<code>` receives each key–value pair in the `<property list>` as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the `<key>` and the `<value>` as its three arguments. For that specific task, `\prop_item:Nn` is faster.

`\prop_map_break:` ☆ `\prop_map_break:`

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\prop_map_break:n` ☆ `\prop_map_break:n {⟨code⟩}`

Used to terminate a `\prop_map...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨code⟩* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

26.8 Viewing property lists

`\prop_show:N` `\prop_show:N ⟨property list⟩`

`\prop_show:c`

Displays the entries in the *⟨property list⟩* in the terminal, and specifies its storage type.

Updated: 2021-04-29

<code>\prop_log:N</code>	<code>\prop_log:N</code> $\langle$ <i>property list</i> $\rangle$
<code>\prop_log:c</code>	Writes the entries in the $\langle$ <i>property list</i> $\rangle$ in the log file, and specifies its storage type.

Updated: 2021-04-29

26.9 Scratch property lists

There is no need to include both flat and linked property lists as scratch variables. We arbitrarily pick the older implementation.

<code>\l_tmpa_prop</code>	Scratch “flat” property lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	

<code>\g_tmpa_prop</code>	Scratch “flat” property lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	

26.10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.

Chapter 27

The l3skip module

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

Many functions take *dimension expressions* (“`<dim expr>`”) or *skip expressions* (“`<skip expr>`”) as arguments.

27.1 Creating and initializing dim variables

<code>\dim_new:N</code>	<code>\dim_new:N <dimension></code>	
<code>\dim_new:c</code>		Creates a new <code><dimension></code> or raises an error if the name is already taken. The declaration is global. The <code><dimension></code> is initially equal to 0pt.
<code>\dim_const:Nn</code>	<code>\dim_const:Nn <dimension> {<dim expr>}</code>	
<code>\dim_const:cn</code>		Creates a new constant <code><dimension></code> or raises an error if the name is already taken. The value of the <code><dimension></code> is set globally to the <code><dim expr></code> .
<code>\dim_zero:N</code>	<code>\dim_zero:N <dimension></code>	
<code>\dim_zero:c</code>		Sets <code><dimension></code> to 0pt.
<code>\dim_gzero:N</code>		
<code>\dim_gzero:c</code>		
<code>\dim_zero_new:N</code>	<code>\dim_zero_new:N <dimension></code>	
<code>\dim_zero_new:c</code>		Ensures that the <code><dimension></code> exists globally by applying <code>\dim_new:N</code> if necessary, then
<code>\dim_gzero_new:N</code>		applies <code>\dim_(g)zero:N</code> to leave the <code><dimension></code> set to zero.
<code>\dim_gzero_new:c</code>		

<code>\dim_if_exist_p:N</code>	★	<code>\dim_if_exist_p:N</code>	$\langle dimension \rangle$
<code>\dim_if_exist_p:c</code>	★	<code>\dim_if_exist:NTF</code>	$\langle dimension \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\dim_if_exist:NTF</code>	★	Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.	
<code>\dim_if_exist:cTF</code>	★		

27.2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn</code>	$\langle dimension \rangle$ $\{\langle dim\ expr \rangle\}$
<code>\dim_add:cn</code>	Adds the result of the $\langle dim\ expr \rangle$ to the current content of the $\langle dimension \rangle$.	
<code>\dim_gadd:Nn</code>		
<code>\dim_gadd:cn</code>		

<code>\dim_set:Nn</code>	<code>\dim_set:Nn</code>	$\langle dimension \rangle$ $\{\langle dim\ expr \rangle\}$
<code>\dim_set:(cn NV cV)</code>	Sets $\langle dimension \rangle$ to the value of $\langle dim\ expr \rangle$, which must evaluate to a length with units.	
<code>\dim_gset:Nn</code>		
<code>\dim_gset:(cn NV cV)</code>		

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN</code>	$\langle dimension_1 \rangle$ $\langle dimension_2 \rangle$
<code>\dim_set_eq:(cn Nc cc)</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.	
<code>\dim_gset_eq:NN</code>		
<code>\dim_gset_eq:(cn Nc cc)</code>		

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn</code>	$\langle dimension \rangle$ $\{\langle dim\ expr \rangle\}$
<code>\dim_sub:cn</code>	Subtracts the result of the $\langle dim\ expr \rangle$ from the current content of the $\langle dimension \rangle$.	
<code>\dim_gsub:Nn</code>		
<code>\dim_gsub:cn</code>		

27.3 Utilities for dimension calculations

<code>\dim_abs:n</code>	★	<code>\dim_abs:n</code>	$\{\langle dim\ expr \rangle\}$
Converts the $\langle dim\ expr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.			

<code>\dim_max:nn</code>	★	<code>\dim_max:nn</code>	$\{\langle dim\ expr_1 \rangle\}$ $\{\langle dim\ expr_2 \rangle\}$
<code>\dim_min:nn</code>	★	<code>\dim_min:nn</code>	$\{\langle dim\ expr_1 \rangle\}$ $\{\langle dim\ expr_2 \rangle\}$
Evaluates the two $\langle dim\ exprs \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.			

<code>\dim_ratio:nn</code>	☆ <code>\dim_ratio:nn {⟨dim expr₁⟩} {⟨dim expr₂⟩}</code>

Parses the two *⟨dim exprs⟩* and converts the ratio of the two to a form suitable for use inside a *⟨dim expr⟩*. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Ne \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

27.4 Dimension expression conditionals

<code>\dim_compare_p:nNn</code>	★ <code>\dim_compare_p:nNn {⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩}</code>
<code>\dim_compare_p:(vNn nNv vNv)</code>	★ <code>\dim_compare:nNnTF</code>
<code>\dim_compare:nNnTF</code>	★ <code>{⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩}</code>
<code>\dim_compare:(vNn nNv vNv)TF</code>	★ <code>{⟨true code⟩} {⟨false code⟩}</code>

This function first evaluates each of the *⟨dim exprs⟩* as described for `\dim_eval:n`. The two results are then compared using the *⟨relation⟩*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

Variables may be used directly in the *⟨dim exprs⟩* here: as a result, there is no need for V- (or e-) type variants. The v-type variants are provided for convenience.

```

\dim_compare_p:n * \dim_compare_p:n
\dim_compare:nTF * {
    <dim expr1> <relation1>
    ...
    <dim exprN> <relationN>
    <dim exprN+1>
}
\dim_compare:nTF
{
    <dim expr1> <relation1>
    ...
    <dim exprN> <relationN>
    <dim exprN+1>
}
{<true code>} {<false code>}

```

This function evaluates the $\langle \text{dim exprs} \rangle$ as described for $\backslash\text{dim\_eval:n}$ and compares consecutive result using the corresponding $\langle \text{relation} \rangle$, namely it compares $\langle \text{dim expr}_1 \rangle$ and $\langle \text{dim expr}_2 \rangle$ using the $\langle \text{relation}_1 \rangle$, then $\langle \text{dim expr}_2 \rangle$ and $\langle \text{dim expr}_3 \rangle$ using the $\langle \text{relation}_2 \rangle$, until finally comparing $\langle \text{dim expr}_N \rangle$ and $\langle \text{dim expr}_{N+1} \rangle$ using the $\langle \text{relation}_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle \text{dim expr} \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle \text{dim expr} \rangle$ is evaluated and no other comparison is performed. The $\langle \text{relations} \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than $\backslash\text{dim\_compare:nNnTF}$ but around 5 times slower.

```

\dim_case:nn  ☆ \dim_case:nnTF {⟨test dim expr⟩}
\dim_case:nnTF ☆ {
    {⟨dim expr case1⟩} {⟨code case1⟩}
    {⟨dim expr case2⟩} {⟨code case2⟩}
    ...
    {⟨dim expr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}

```

This function evaluates the $\langle \text{test dim expr} \rangle$ and compares this in turn to each of the $\langle \text{dim expr case} \rangle$ s until a match is found. If the two are equal then the associated $\langle \text{code} \rangle$ is left in the input stream and other cases are discarded. If any of the cases are matched, the $\langle \text{true code} \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle \text{false code} \rangle$ is inserted. The function $\backslash\text{dim\_case:nn}$, which does nothing if there is no match, is also available. For example

```

\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
    { 5 pt }      { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }   { Negative }
}
{ No idea! }

```

leaves “Medium” in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

27.5 Dimension expression loops

```

\dim_do_until:nNnn ☆ \dim_do_until:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}

```

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle \text{dim exprs} \rangle$ as described for $\backslash\text{dim\_compare:nNnTF}$. If the test is **false** then the $\langle \text{code} \rangle$ is inserted into the input stream again and a loop occurs until the $\langle \text{relation} \rangle$ is **true**.

```

\dim_do_while:nNnn ☆ \dim_do_while:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}

```

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle \text{dim exprs} \rangle$ as described for $\backslash\text{dim\_compare:nNnTF}$. If the test is **true** then the $\langle \text{code} \rangle$ is inserted into the input stream again and a loop occurs until the $\langle \text{relation} \rangle$ is **false**.

<code>\dim_until_do:nNnn</code> ☆	<code>\dim_until_do:nNnn {<dim expr₁>} <relation> {<dim expr₂>} {<code>}</code>
	Evaluates the relationship between the two <code><dim exprs></code> as described for <code>\dim_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is false. After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is true.
<code>\dim_while_do:nNnn</code> ☆	<code>\dim_while_do:nNnn {<dim expr₁>} <relation> {<dim expr₂>} {<code>}</code>
	Evaluates the relationship between the two <code><dim exprs></code> as described for <code>\dim_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is true. After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is false.
<code>\dim_do_until:nn</code> ☆	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the <code><dimension relation></code> as described for <code>\dim_compare:nTF</code> . If the test is false then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is true.
<code>\dim_do_while:nn</code> ☆	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the <code><dimension relation></code> as described for <code>\dim_compare:nTF</code> . If the test is true then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is false.
<code>\dim_until_do:nn</code> ☆	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <code><dimension relation></code> as described for <code>\dim_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is false. After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is true.
<code>\dim_while_do:nn</code> ☆	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <code><dimension relation></code> as described for <code>\dim_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is true. After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is false.

27.6 Dimension step functions

<code>\dim_step_function:nnnN</code> ☆	<code>\dim_step_function:nnnN {<initial value>} {<step>} {<final value>} <function></code>
	This function first evaluates the <code><initial value></code> , <code><step></code> and <code><final value></code> , all of which should be dimension expressions. The <code><function></code> is then placed in front of each <code><value></code> from the <code><initial value></code> to the <code><final value></code> in turn (using <code><step></code> between each <code><value></code>). The <code><step></code> must be non-zero. If the <code><step></code> is positive, the loop stops when the <code><value></code> becomes larger than the <code><final value></code> . If the <code><step></code> is negative, the loop stops when the <code><value></code> becomes smaller than the <code><final value></code> . The <code><function></code> should absorb one argument.

`\dim_step_inline:nnnn` `\dim_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}`

This function first evaluates the `<initial value>`, `<step>` and `<final value>`, all of which should be dimension expressions. Then for each `<value>` from the `<initial value>` to the `<final value>` in turn (using `<step>` between each `<value>`), the `<code>` is inserted into the input stream with `#1` replaced by the current `<value>`. Thus the `<code>` should define a function of one argument (`#1`).

`\dim_step_variable:nnnNn` `\dim_step_variable:nnnNn {<initial value>} {<step>} {<final value>} <tl var> {<code>}`

This function first evaluates the `<initial value>`, `<step>` and `<final value>`, all of which should be dimension expressions. Then for each `<value>` from the `<initial value>` to the `<final value>` in turn (using `<step>` between each `<value>`), the `<code>` is inserted into the input stream, with the `<tl var>` defined as the current `<value>`. Thus the `<code>` should make use of the `<tl var>`.

27.7 Using dim expressions and variables

`\dim_eval:n *` `\dim_eval:n {<dim expr>}`

Evaluates the `<dim expr>`, expanding any dimensions and token list variables within the `<expression>` to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a `<dimension denotation>` after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a $\TeX$ -style assignment as it is *not* an `<internal dimension>`.

`\dim_sign:n *` `\dim_sign:n {<dim expr>}`

Evaluates the `<dim expr>` then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\dim_use:N *` `\dim_use:N <dimension>`

`\dim_use:c *`

Recovers the content of a `<dimension>` and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a `<dimension>` is required (such as in the argument of `\dim_eval:n`).

$\TeX$ hackers note: `\dim_use:N` is the $\TeX$ primitive `\the`: this is one of several $\LaTeX$ 3 names for this primitive.

`\dim_to_decimal:n *` `\dim_to_decimal:n {<dim expr>}`

Evaluates the `<dim expr>`, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by $\TeX$ to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves 1.00374 in the input stream, i.e., the magnitude of one “big point” when converted to ($\TeX$) points.

`\dim_to_decimal_in_bp:n` ★ `\dim_to_decimal_in_bp:n {⟨dim expr⟩}`

Updated: 2023-05-20

Evaluates the `⟨dim expr⟩`, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by `TeX` to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, i.e., the magnitude of one (`TeX`) point when converted to big points.

TeXhackers note: The implementation of this function is re-entrant: the result of

```
\dim_compare:nNnTF
{ <x>bp } =
{ \dim_to_decimal_in_bp:n { <x>bp } bp }
```

will be logically `true`. The decimal representations may differ provided they produce the same `TeX` dimension.

`\dim_to_decimal_in_cc:n` ★ `\dim_to_decimal_in_cm:n` `{⟨dim expr⟩}`

`\dim_to_decimal_in_cm:n` ★

`\dim_to_decimal_in_dd:n` ★

`\dim_to_decimal_in_in:n` ★

`\dim_to_decimal_in_mm:n` ★

`\dim_to_decimal_in_pc:n` ★

New: 2023-05-20

Evaluates the `⟨dim expr⟩`, and leaves the result, expressed with the appropriate scaling in the input stream, with *no units*. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

The maximum `TeX` allowable dimension value (available as `\maxdimen` in plain `TeX` and `LaTeX` and `\c_max_dim` in `expl3`) can only be expressed exactly in the units `pt`, `bp` and `sp`. The maximum allowable input values to five decimal places are

```
1276.00215 cc
575.83174 cm
15312.02584 dd
226.70540 in
5758.31742 mm
1365.33333 pc
```

(Note that these are not all equal, but rather any larger value will overflow due to the way `TeX` converts to `sp`.) Values given to five decimal places larger than these will result in `TeX` errors; the behavior if additional decimal places are given depends on the `TeX` internals and thus larger values are *not* supported by `expl3`.

TeXhackers note: The implementation of these functions is re-entrant: the result of

```
\dim_compare:nNnTF
{ <x><unit> } =
{ \dim_to_decimal_in_<unit>:n { <x><unit> } <unit> }
```

will be logically `true`. The decimal representations may differ provided they produce the same `TeX` dimension.

`\dim_to_decimal_in_sp:n` ★ `\dim_to_decimal_in_sp:n {⟨dim expr⟩}`

Evaluates the `⟨dim expr⟩`, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result is necessarily an integer.

`\dim_to_decimal_in_unit:nn` ★ `\dim_to_decimal_in_unit:nn {⟨dim expr1⟩} {⟨dim expr2⟩}`

Updated: 2023-05-20

Evaluates the `⟨dim exprs⟩`, and leaves the value of `⟨dim expr1⟩`, expressed in a unit given by `⟨dim expr2⟩`, in the input stream. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35278 in the input stream, i.e., the magnitude of one big point when expressed in millimetres. The conversions do *not* guarantee that `TEX` would yield identical results for the direct input in an equality test, thus for instance

```
\dim_compare:nNnTF
{ 1bp } =
{ \dim_to_decimal_in_unit:nn { 1bp } { 1mm } mm }
```

will take the `false` branch.

`\dim_to_fp:n` ★ `\dim_to_fp:n {⟨dim expr⟩}`

Expands to an internal floating point number equal to the value of the `⟨dim expr⟩` in `pt`. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

27.8 Viewing dim variables

`\dim_show:N` `\dim_show:N ⟨dimension⟩`

`\dim_show:c` Displays the value of the `⟨dimension⟩` on the terminal.

`\dim_show:n` `\dim_show:n {⟨dim expr⟩}`

Displays the result of evaluating the `⟨dim expr⟩` on the terminal.

`\dim_log:N` `\dim_log:N ⟨dimension⟩`

`\dim_log:c` Writes the value of the `⟨dimension⟩` in the log file.

`\dim_log:n` `\dim_log:n {⟨dim expr⟩}`

Writes the result of evaluating the `⟨dim expr⟩` in the log file.

27.9 Constant dimensions

`\c_max_dim` The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

`\c_zero_dim` A zero length as a dimension. This can also be used as a component of a skip.

27.10 Scratch dimensions

`\l_tmpa_dim` Scratch dimension for local assignment. These are never used by the kernel code, and so
`\l_tmpb_dim` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_dim` Scratch dimension for global assignment. These are never used by the kernel code, and
`\g_tmpb_dim` so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

27.11 Inserting dimensions into the output

`\dim_horizontal:N` `\dim_horizontal:N` $\langle dim \rangle$
`\dim_horizontal:c` `\dim_horizontal:n` $\{\langle dim \text{ expr} \rangle\}$
`\dim_horizontal:n` Inserts a horizontal $\langle dim \rangle$ into the current list.

New: 2026-02-01

T_EXhackers note: `\dim_horizontal:N` is a wrapper around the T_EX primitive `\kern`.

`\dim_vertical:N` `\dim_vertical:N` $\langle dim \rangle$
`\dim_vertical:c` `\dim_vertical:n` $\{\langle dim \text{ expr} \rangle\}$
`\dim_vertical:n` Inserts a vertical $\langle dim \rangle$ into the current list.

New: 2026-02-01

T_EXhackers note: `\dim_vertical:N` is a wrapper around the T_EX primitive `\kern`.

27.12 Creating and initializing skip variables

`\skip_new:N` `\skip_new:N` $\langle skip \rangle$
`\skip_new:c` Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ is initially equal to 0 pt.

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {<skip expr>}</code>
<code>\skip_const:cn</code>	Creates a new constant <code><skip></code> or raises an error if the name is already taken. The value of the <code><skip></code> is set globally to the <code><skip expr></code> .
<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets <code><skip></code> to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	
<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N <skip></code>
<code>\skip_zero_new:c</code>	Ensures that the <code><skip></code> exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the <code><skip></code> set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	
<code>\skip_if_exist_p:N *</code>	<code>\skip_if_exist_p:N <skip></code>
<code>\skip_if_exist_p:c *</code>	<code>\skip_if_exist:NTF <skip> {<true code>} {<false code>}</code>
<code>\skip_if_exist:NTF *</code>	Tests whether the <code><skip></code> is currently defined. This does not check that the <code><skip></code> really is a skip variable.
<code>\skip_if_exist:cTF *</code>	

27.13 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {<skip expr>}</code>
<code>\skip_add:cn</code>	Adds the result of the <code><skip expr></code> to the current content of the <code><skip></code> .
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	
<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {<skip expr>}</code>
<code>\skip_set:(cn NV cV)</code>	Sets <code><skip></code> to the value of <code><skip expr></code> , which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:(cn NV cV)</code>	
<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN <skip₁₂</code>
<code>\skip_set_eq:(cn Nc cc)</code>	Sets the content of <code><skip_{1 equal to that of <code><skip_{2.}</code>}</code>
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cn Nc cc)</code>	
<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expr>}</code>
<code>\skip_sub:cn</code>	Subtracts the result of the <code><skip expr></code> from the current content of the <code><skip></code> .
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	

27.14 Skip expression conditionals

```

\skip_if_eq_p:nn * \skip_if_eq_p:nn {\skip expr1} {\skip expr2}
\skip_if_eq:nnTF * \skip_if_eq:nnTF
                    {\skip expr1} {\skip expr2}
                    {\true code} {\false code}

```

This function first evaluates each of the $\langle skip\ exprs \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, i.e., both the fixed and rubber components must be the same for the test to be true.

```

\skip_if_finite_p:n * \skip_if_finite_p:n {\skip expr}
\skip_if_finite:nTF * \skip_if_finite:nTF {\skip expr} {\true code} {\false code}

```

Evaluates the $\langle skip\ expr \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

27.15 Using skip expressions and variables

```

\skip_eval:n * \skip_eval:n {\skip expr}

```

Evaluates the $\langle skip\ expr \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue\ denotation \rangle$ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal\ glue \rangle$.

```

\skip_use:N * \skip_use:N \skip
\skip_use:c *

```

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ or $\langle skip \rangle$ is required (such as in the argument of `\skip_eval:n`).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

27.16 Viewing skip variables

```

\skip_show:N * \skip_show:N \skip
\skip_show:c *

```

Displays the value of the $\langle skip \rangle$ on the terminal.

```

\skip_show:n * \skip_show:n {\skip expr}

```

Displays the result of evaluating the $\langle skip\ expr \rangle$ on the terminal.

<code>\skip_log:N</code>	<code>\skip_log:N</code>	$\langle skip \rangle$
<code>\skip_log:c</code>	Writes the value of the $\langle skip \rangle$ in the log file.	

<code>\skip_log:n</code>	<code>\skip_log:n</code>	$\{\langle skip \ expr \rangle\}$
Writes the result of evaluating the $\langle skip \ expr \rangle$ in the log file.		

27.17 Constant skips

<code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.

<code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.

27.18 Scratch skips

<code>\l_tmpa_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_skip</code>	

<code>\g_tmpa_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_skip</code>	

27.19 Inserting skips into the output

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N</code>	$\langle skip \rangle$
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:n</code> $\{\langle skip \ expr \rangle\}$	
<code>\skip_horizontal:n</code>	Inserts a horizontal $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.	

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip`.

<code>\skip_vertical:N</code>	<code>\skip_vertical:N</code>	$\langle skip \rangle$
<code>\skip_vertical:c</code>	<code>\skip_vertical:n</code> $\{\langle skip \ expr \rangle\}$	
<code>\skip_vertical:n</code>	Inserts a vertical $\langle skip \rangle$ into the current list. The argument can also be a $\langle dim \rangle$.	

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip`.

27.20 Creating and initializing muskip variables

<code>\muskip_new:N</code>	<code>\muskip_new:N <muskip></code>
<code>\muskip_new:c</code>	Creates a new <code><muskip></code> or raises an error if the name is already taken. The declaration is global. The <code><muskip></code> is initially equal to 0 mu.
<code>\muskip_const:Nn</code>	<code>\muskip_const:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_const:cn</code>	Creates a new constant <code><muskip></code> or raises an error if the name is already taken. The value of the <code><muskip></code> is set globally to the <code><muskip expr></code> .
<code>\muskip_zero:N</code>	<code>\skip_zero:N <muskip></code>
<code>\muskip_zero:c</code>	Sets <code><muskip></code> to 0 mu.
<code>\muskip_gzero:N</code>	
<code>\muskip_gzero:c</code>	
<code>\muskip_zero_new:N</code>	<code>\muskip_zero_new:N <muskip></code>
<code>\muskip_zero_new:c</code>	Ensures that the <code><muskip></code> exists globally by applying <code>\muskip_new:N</code> if necessary, then applies <code>\muskip_(g)zero:N</code> to leave the <code><muskip></code> set to zero.
<code>\muskip_gzero_new:N</code>	
<code>\muskip_gzero_new:c</code>	
<code>\muskip_if_exist_p:N</code>	<code>\muskip_if_exist_p:N <muskip></code>
<code>\muskip_if_exist_p:c</code>	<code>\muskip_if_exist:NTF <muskip> {<true code>} {<false code>}</code>
<code>\muskip_if_exist:NTF</code>	Tests whether the <code><muskip></code> is currently defined. This does not check that the <code><muskip></code> really is a muskip variable.
<code>\muskip_if_exist:cTF</code>	

27.21 Setting muskip variables

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_add:cn</code>	Adds the result of the <code><muskip expr></code> to the current content of the <code><muskip></code> .
<code>\muskip_gadd:Nn</code>	
<code>\muskip_gadd:cn</code>	
<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_set:(cn NV cV)</code>	Sets <code><muskip></code> to the value of <code><muskip expr></code> , which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:Nn</code>	
<code>\muskip_gset:(cn NV cV)</code>	
<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁₂</code>
<code>\muskip_set_eq:(cN Nc cc)</code>	Sets the content of <code><muskip_{1 equal to that of <code><muskip_{2.}</code>}</code>
<code>\muskip_gset_eq:NN</code>	
<code>\muskip_gset_eq:(cN Nc cc)</code>	
<code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expr>}</code>
<code>\muskip_sub:cn</code>	Subtracts the result of the <code><muskip expr></code> from the current content of the <code><muskip></code> .
<code>\muskip_gsub:Nn</code>	
<code>\muskip_gsub:cn</code>	

27.22 Using muskip expressions and variables

`\muskip_eval:n` ★ `\muskip_eval:n {⟨muskip expr⟩}`

Evaluates the `⟨muskip expr⟩`, expanding any skips and token list variables within the `⟨expression⟩` to their content (without requiring `\muskip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a `⟨muglue denotation⟩` after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TeX-style assignment as it is *not* an `⟨internal muglue⟩`.

`\muskip_use:N` ★ `\muskip_use:N ⟨muskip⟩`

`\muskip_use:c` ★ Recovers the content of a `⟨skip⟩` and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a `⟨dimension⟩` is required (such as in the argument of `\muskip_eval:n`).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

27.23 Viewing muskip variables

`\muskip_show:N` `\muskip_show:N ⟨muskip⟩`

`\muskip_show:c` Displays the value of the `⟨muskip⟩` on the terminal.

`\muskip_show:n` `\muskip_show:n {⟨muskip expr⟩}`

Displays the result of evaluating the `⟨muskip expr⟩` on the terminal.

`\muskip_log:N` `\muskip_log:N ⟨muskip⟩`

`\muskip_log:c` Writes the value of the `⟨muskip⟩` in the log file.

`\muskip_log:n` `\muskip_log:n {⟨muskip expr⟩}`

Writes the result of evaluating the `⟨muskip expr⟩` in the log file.

27.24 Constant muskips

`\c_max_muskip`

The maximum value that can be stored as a muskip, with no stretch nor shrink component.

`\c_zero_muskip`

A zero length as a muskip, with no stretch nor shrink component.

27.25 Scratch muskips

<code>\l_tmpa_muskip</code>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_muskip</code>	

<code>\g_tmpa_muskip</code>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_muskip</code>	

27.26 Primitive conditional

<code>\if_dim:w</code>	<code>\star</code>	<code>\if_dim:w</code>	<code><dimen₁></code>	<code><relation></code>	<code><dimen₂></code>
			<code><true code></code>		
			<code>\else:</code>		
			<code><false></code>		
			<code>\fi:</code>		

Compare two dimensions. The `<relation>` is one of `<`, `=` or `>` with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

Chapter 28

The l3keys module

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behavior. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimizing the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n    = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

As illustrated, keys are created inside a $\langle module \rangle$: a set of related keys, typically those for a single module/L^AT_EX 2_ε package. See Section 28.2 for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
{ \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
  \group_begin:
    \keys_set:nn { mymodule } { #1 }
    % Main code for \MyModuleMacro
  \group_end:
}
```

Key names may contain any tokens except `:`, as they are handled internally using `\tl_to_str:n`. As discussed in section 28.2, it is suggested that the character `/` is reserved for sub-division of keys into different subsets. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

28.1 Creating keys

```
\keys_define:nn \keys_define:nn { $\langle module \rangle$ } { $\langle keyval list \rangle$ }
```

Parses the $\langle keyval list \rangle$ and defines the keys listed there for $\langle module \rangle$. The $\langle module \rangle$ name is treated as a string. In practice the $\langle module \rangle$ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The $\langle keyval list \rangle$ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
  keyname .code:n = Some-code~using~#1,
  keyname .value_required:n = true
}
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary $\langle key \rangle$, which when used may be supplied with a $\langle value \rangle$. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, etc., override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviors may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some~code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
  keyname .value_required:n = true,
  keyname .code:n          = Some~code~using~#1
}
```

Note that all key properties define the key within the current $\text{\TeX}$ group, with an exception that the special `.undefine:` property *undefines* the key within the current $\text{\TeX}$ group.

<code>.bool_set:N</code> <code>.bool_set:c</code> <code>.bool_gset:N</code> <code>.bool_gset:c</code>	$\langle key \rangle$ <code>.bool_set:N</code> = $\langle boolean \rangle$ Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$. If the $\langle value \rangle$ is given, it must be one either “true” or “false”; it may be omitted, which is equivalent to true. If the variable does not exist, it will be created globally at the point that the key is set up.

<code>.bool_set_inverse:N</code> <code>.bool_set_inverse:c</code> <code>.bool_gset_inverse:N</code> <code>.bool_gset_inverse:c</code>	$\langle key \rangle$ <code>.bool_set_inverse:N</code> = $\langle boolean \rangle$ Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either “true” or “false”). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

<code>.choice:</code>	$\langle key \rangle$ <code>.choice:</code> Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 28.3.

<code>.choices:nn</code> <code>.choices:(Vn en on)</code>	$\langle key \rangle$ <code>.choices:nn</code> = $\{\langle choices \rangle\} \{\langle code \rangle\}$ Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, <code>\l_keys_choice_str</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 28.3.

<code>.clist_set:N</code> <code>.clist_set:c</code> <code>.clist_gset:N</code> <code>.clist_gset:c</code>	$\langle key \rangle$ <code>.clist_set:N</code> = $\langle comma list variable \rangle$ Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

`.code:n` $\langle key \rangle$ `.code:n = {\code}`

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (#1), which will be the $\langle value \rangle$ given for the $\langle key \rangle$.

`.cs_set:Np` $\langle key \rangle$ `.cs_set:Np = \control sequence \arg. spec.`

`.cs_set:cp`

`.cs_set_protected:Np` Defines $\langle key \rangle$ to set $\langle control\ sequence \rangle$ to have $\langle arg.\ spec. \rangle$ and replacement text $\langle value \rangle$.

`.cs_set_protected:cp`

`.cs_gset:Np`

`.cs_gset:cp`

`.cs_gset_protected:Np`

`.cs_gset_protected:cp`

`.default:n` $\langle key \rangle$ `.default:n = {\default}`

`.default:(V|e|o)`

Creates a $\langle default \rangle$ value for $\langle key \rangle$, which is used if no value is given. This will be used if only the key name is given, but not if a blank $\langle value \rangle$ is given:

```
\keys_define:nn { mymodule }
{
  key .code:n      = Hello~#1,
  key .default:n = World
}
\keys_set:nn { mymodule }
{
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,    % Prints 'Hello '
}
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

When no value is given for a key as part of `\keys_set:nn`, the `.default:n` value provides the value before key properties are considered. The only exception is when the `.value_required:n` property is active: a required value cannot be supplied by the default, and must be explicitly given as part of `\keys_set:nn`.

`.dim_set:N` $\langle key \rangle$ `.dim_set:N = \dimension`

`.dim_set:c`

`.dim_gset:N`

`.dim_gset:c`

Defines $\langle key \rangle$ to set $\langle dimension \rangle$ to $\langle value \rangle$ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.fp_set:N` $\langle key \rangle$ `.fp_set:N = \fp var`

`.fp_set:c`

`.fp_gset:N`

`.fp_gset:c`

Defines $\langle key \rangle$ to set $\langle fp\ var \rangle$ to $\langle value \rangle$ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.groups:n $\langle key \rangle$.groups:n = { $\langle groups \rangle$ }

Defines $\langle key \rangle$ as belonging to the $\langle groups \rangle$ (a comma-separated list). Groups provide a “secondary axis” for selectively setting keys, and are described in Section 28.7.

TeXhackers note: The $\langle groups \rangle$ argument is turned into a string then interpreted as a comma-separated list, so group names cannot contain commas nor start or end with a space character.

.inherit:n $\langle key \rangle$.inherit:n = { $\langle parents \rangle$ }

Specifies that the $\langle key \rangle$ path should inherit the keys listed as any of the $\langle parents \rangle$ (a comma list), which can be a module or a sub-division thereof. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the $\langle parent \rangle$ after the use of .inherit:n and will be active. If more than one $\langle parent \rangle$ is specified, the presence of the $\langle key \rangle$ will be tested for each in turn, with the first successful hit taking priority.

.initial:n $\langle key \rangle$.initial:n = { $\langle value \rangle$ }

.initial:(V|e|o)

Initializes the $\langle key \rangle$ with the $\langle value \rangle$, equivalent to

```
\keys_set:nn { $\langle module \rangle$ } {  $\langle key \rangle$  =  $\langle value \rangle$  }
```

.int_set:N $\langle key \rangle$.int_set:N = $\langle integer \rangle$

.int_set:c

.int_gset:N

.int_gset:c

Defines $\langle key \rangle$ to set $\langle integer \rangle$ to $\langle value \rangle$ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.legacy_if_set:n $\langle key \rangle$.legacy_if_set:n = $\langle switch \rangle$

.legacy_if_gset:n

.legacy_if_set_inverse:n

.legacy_if_gset_inverse:n

Defines $\langle key \rangle$ to set legacy \if $\langle switch \rangle$ to $\langle value \rangle$ (which must be either “true” or “false”). The $\langle switch \rangle$ is the name of the switch *without the leading if*.

The inverse versions will set the $\langle switch \rangle$ to the logical opposite of the $\langle value \rangle$.

Updated: 2022-01-15

.meta:n $\langle key \rangle$.meta:n = { $\langle keyval list \rangle$ }

Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go. The $\langle keyval list \rangle$ can refer as #1 to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$ ’s default value).

.meta:nn $\langle key \rangle$.meta:nn = { $\langle path \rangle$ } { $\langle keyval list \rangle$ }

Makes $\langle key \rangle$ a meta-key, which will set $\langle keyval list \rangle$ in one go using the $\langle path \rangle$ in place of the current one. The $\langle keyval list \rangle$ can refer as #1 to the value given at the time the $\langle key \rangle$ is used (or, if no value is given, the $\langle key \rangle$'s default value).

.multichoice: $\langle key \rangle$.multichoice:

Sets $\langle key \rangle$ to act as a multiple choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 28.3.

.multichoices:nn $\langle key \rangle$.multichoices:nn { $\langle choices \rangle$ } { $\langle code \rangle$ }

.multichoices:(Vn|en|on)

Sets $\langle key \rangle$ to act as a multiple choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, $\backslash l\_keys\_choice\_str$ will be the name of the choice made, and $\backslash l\_keys\_choice\_int$ will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 28.3.

.muskip_set:N $\langle key \rangle$.muskip_set:N = $\langle muskip \rangle$

.muskip_set:c

.muskip_gset:N

.muskip_gset:c

Defines $\langle key \rangle$ to set $\langle muskip \rangle$ to $\langle value \rangle$ (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.prop_put:N $\langle key \rangle$.prop_put:N = $\langle property list \rangle$

.prop_put:c

.prop_gput:N

.prop_gput:c

Defines $\langle key \rangle$ to put the $\langle value \rangle$ onto the $\langle property list \rangle$ stored under the $\langle key \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

.skip_set:N $\langle key \rangle$.skip_set:N = $\langle skip \rangle$

.skip_set:c

.skip_gset:N

.skip_gset:c

Defines $\langle key \rangle$ to set $\langle skip \rangle$ to $\langle value \rangle$ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.str_set:N $\langle key \rangle$.str_set:N = $\langle string variable \rangle$

.str_set:c

.str_gset:N

.str_gset:c

Defines $\langle key \rangle$ to set $\langle string variable \rangle$ to $\langle value \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

New: 2021-10-30

.str_set_e:N $\langle key \rangle$.str_set_e:N = $\langle string variable \rangle$

.str_set_e:c

.str_gset_e:N

.str_gset_e:c

Defines $\langle key \rangle$ to set $\langle string variable \rangle$ to $\langle value \rangle$, which will be subjected to an e-type expansion (i.e., using $\backslash str\_set:Ne$). If the variable does not exist, it is created globally at the point that the key is set up.

New: 2023-09-18

.tl_set:N $\langle key \rangle$.tl_set:N = $\langle tl var \rangle$

.tl_set:c

.tl_gset:N

.tl_gset:c

Defines $\langle key \rangle$ to set $\langle tl var \rangle$ to $\langle value \rangle$. If the variable does not exist, it is created globally at the point that the key is set up.

`.tl_set_e:N` $\langle key \rangle$ `.tl_set_e:N =  $\langle tl\ var \rangle$`

`.tl_set_e:c` Defines $\langle key \rangle$ to set $\langle tl\ var \rangle$ to $\langle value \rangle$, which will be subjected to an e-type expansion (i.e., using `\tl_set:Ne`). If the variable does not exist, it is created globally at the point that the key is set up.

New: 2023-09-18

`.undefine:` $\langle key \rangle$ `.undefine:`

Removes the definition of the $\langle key \rangle$ within the current T_EX group.

`.value_forbidden:n` $\langle key \rangle$ `.value_forbidden:n = true|false`

Specifies that $\langle key \rangle$ cannot receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is given then an error will be issued. Setting the property “false” cancels the restriction.

`.value_required:n` $\langle key \rangle$ `.value_required:n = true|false`

Specifies that $\langle key \rangle$ must receive a $\langle value \rangle$ when used. If a $\langle value \rangle$ is not given then an error will be issued. Setting the property “false” cancels the restriction.

28.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several subsets for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subset }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subset / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subset/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

28.3 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependent only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\str_use:N \l_keys_choice_str',~
      which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

<code>\l_keys_choice_int</code> <code>\l_keys_choice_str</code>	Inside the code block for a choice generated using <code>.choices:nn</code> , the variables <code>\l_keys_choice_str</code> and <code>\l_keys_choice_int</code> are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using <code>.code:n</code> , the value passed to the key (i.e., the choice name) is also available as <code>#1</code> .

Updated: 2025-09-29

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_str` or `\l_keys_choice_int`. These variables do not have defined behavior when used outside of code created using `.choices:nn` (i.e., anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special **unknown** choice. The general behavior of the **unknown** key is described in Section 28.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
```

```

key / unknown .code:n =
\msg_error:nneee { mymodule } { unknown-choice }
{ key } % Name of choice key
{ choice-a , choice-b , choice-c } % Valid choices
{ \exp_not:n {#1} } % Invalid choice given
}

```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

```

\keys_define:nn { mymodule }
{
  key .multichoices:nn =
  { choice-a, choice-b, choice-c }
  {
    You~gave~choice~'\str_use:N \l_keys_choice_str',~
    which~is~in~position~
    \int_use:N \l_keys_choice_int \c_space_tl
    in~the~list.
  }
}

```

and

```

\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid.

When a multiple choice key is set

```

\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```

\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}

```

Thus each separate choice will have passed to it the `\l_keys_choice_str` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

28.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, `l3keys` allows this information to be specified using the `.usage:n` property.

<code>.usage:n</code>	$\langle key \rangle$ <code>.usage:n = $\langle scope \rangle$</code>
New: 2022-01-10	Defines the $\langle key \rangle$ to have usage within the $\langle scope \rangle$, which should be one of general , preamble or load .

<code>\l_keys_usage_load_prop</code>
<code>\l_keys_usage_preamble_prop</code>
New: 2022-01-10

`l3keys` itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

28.5 Setting keys

<code>\keys_set:nn</code>	<code>\keys_set:nn {$\langle module \rangle$} {$\langle keyval list \rangle$}</code>
<code>\keys_set:(nV nv ne no)</code>	Parses the $\langle keyval list \rangle$, and sets those keys which are defined for $\langle module \rangle$. The behavior on finding an unknown key can be set by defining a special unknown key: this is illustrated later.

<code>\l_keys_path_str</code>	For each key processed, information of the full <i>path</i> of the key, the <i>name</i> of the key and the <i>value</i> of the key is available within two string and one token list variables. These may be used within the code of the key.
<code>\l_keys_key_str</code>	
<code>\l_keys_value_tl</code>	
Updated: 2020-02-08	The <i>path</i> of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

28.5.1 Expanding the values of keys

To allow the user to apply expansion of values when the key is set, key names can be followed by an expansion specifier. This is given by appending `:` and a single letter specifier to the key name. The latter are the normal argument specifiers for `expl3`, thus may be one of `n` (redundant but supported), `o`, `V`, `v` or `e`, or `N` (again redundant) or `c`. Thus for example

```
\tl_new:N \l__mymodule_value_tl
\dim_new:N \l__mymodule_value_dim
\keys_define:nn { mymodule } { key .code:n = \tl_show:n { "#1" } }
\tl_set:Nn \l__mymodule_value_tl { value }
\dim_set:Nn \l__mymodule_value_dim { 123pt }
\keys_set:nn { mymodule } { key = value }
\keys_set:nn { mymodule } { key:o = \l__mymodule_value_tl }
\keys_set:nn { mymodule } { key:V = \l__mymodule_value_dim }
\keys_set:nn { mymodule }
{
  key:e =
    \l__mymodule_value_tl \c_space_tl
    \dim_use:N \l__mymodule_value_dim
}
```

will result in the output

```
"value"
"value"
"123.0pt"
"value 123.0pt"
```

28.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The `unknown` key also supports the `.default:n` property.

```
\keys_define:nn { mymodule }
{
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_key_str'~to~'#1'. ,
  unknown .default:V = \c_novalue_tl
}
```

Key inheritance does *not* include looking for the special `unknown` key. Handling of `unknown` keys should be set up explicitly for each path where it applies.

28.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using


```

\keys_define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-two   .tl_set:N = \l_my_a_tl           ,
  key-three .tl_set:N = \l_my_b_tl           ,
  key-four  .fp_set:N = \l_my_a_fp           ,
}

```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```

\keys_define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-one   .groups:n = { first }           ,
  key-two   .tl_set:N = \l_my_a_tl           ,
  key-two   .groups:n = { first }           ,
  key-three .tl_set:N = \l_my_b_tl           ,
  key-three .groups:n = { second }          ,
  key-four  .fp_set:N = \l_my_a_fp           ,
}

```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_known:nn</code>	<code>\keys_set_known:nn {<module>} {<keyval list>}</code>
<code>\keys_set_known:(nV nv ne no)</code>	<code>\keys_set_known:nnN {<module>} {<keyval list>} <tl var></code>
<code>\keys_set_known:nnN</code>	<code>\keys_set_known:nnnN {<module>} {<keyval list>} {<root>} <tl var></code>
<code>\keys_set_known:(nVN nvN neN noN)</code>	
<code>\keys_set_known:nnnN</code>	
<code>\keys_set_known:(nVnN nvnN nenN nonN)</code>	

These functions set keys which are known for the `<module>`, and simply ignore other keys. The `\keys_set_known:nn` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser.

In addition, `\keys_set_known:nnN` and `\keys_set_known:nnnN` store the key–value pairs for unknown keys in the `<tl var>` in comma-separated form (i.e., an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	<code>\keys_set_groups:nnnN {<module>} {<groups>} {<keyval list>} <t1 var></code>
<code>\keys_set_groups:nnnN</code>	<code>\keys_set_groups:nnnnN {<module>} {<groups>} {<keyval list>} {<root>}</code>
<code>\keys_set_groups:(nnVN nnvN nnoN)</code>	<code><t1 var></code>
<code>\keys_set_groups:nnnnN</code>	
<code>\keys_set_groups:(nnVnN nnvnN nnonN)</code>	

Updated: 2024-05-08

These functions activate key selection in an “opt-in” sense: only keys assigned to one or more of the `<groups>` specified are set. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

In addition, `\keys_set_groups:nnnN` and `\keys_set_groups:nnnnN` store the key–value pairs for skipped keys in the `<t1 var>` in comma-separated form (i.e., an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_groups:nnnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

<code>\keys_set_exclude_groups:nnn</code>	<code>\keys_set_exclude_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_exclude_groups:(nnV nnv nno)</code>	<code>\keys_set_exclude_groups:nnnN {<module>} {<groups>} {<keyval</code>
<code>\keys_set_exclude_groups:nnnN</code>	<code>list>} <t1 var></code>
<code>\keys_set_exclude_groups:(nnVN nnvN nnoN)</code>	<code>\keys_set_exclude_groups:nnnnN {<module>} {<groups>} {<keyval</code>
<code>\keys_set_exclude_groups:nnnnN</code>	<code>list>} {<root>} <t1 var></code>
<code>\keys_set_exclude_groups:(nnVnN nnvnN nnonN)</code>	

New: 2024-01-10

These functions activate key selection in an “opt-out” sense: keys assigned to one or more of the `<groups>` specified are *not* set. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set.

In addition, `\keys_set_exclude_groups:nnnN` and `\keys_set_exclude_groups:nnnnN` store the key–value pairs for skipped keys in the `<t1 var>` in comma-separated form (i.e., an edited version of the `<keyval list>`). When a `<root>` is given (`\keys_set_exclude_groups:nnnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

28.8 Precompiling keys

<code>\keys_precompile:nnN</code>	<code>\keys_precompile:nnN {<module>} {<keyval list>} <t1 var></code>

New: 2022-03-09

Parses the `<keyval list>` as for `\keys_set:nn`, placing the resulting code for those which set variables or functions into the `<t1 var>`. Thus this function “precompiles” the keyval list into a set of results which can be applied rapidly.

It is important to note that when precompiling keys, no expansion of variables takes place. This means that any key setting which simply stores variable names, rather than variable values, may not work correctly. Most notably, any key setting which uses key status variables (`\l_keys_key_str`, etc.) will yield unpredictable outcomes. As such, keys intended to be precompiled should fully expand any values at the point of setting.

28.9 Utility functions for keys

```
\keys_if_exist_p:nn * \keys_if_exist_p:nn {<module>} {<key>}
\keys_if_exist_p:ne * \keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}
\keys_if_exist:nnTF * Tests if the <key> exists for <module>, i.e., if any code has been defined for <key>.
\keys_if_exist:neTF *
```

Updated: 2022-01-10

```
\keys_if_choice_exist_p:nnn * \keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}
\keys_if_choice_exist:nnnTF * \keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>} {<false code>}
```

Tests if the *<choice>* is defined for the *<key>* within the *<module>*, i.e., if any code has been defined for *<key>/<choice>*. The test is **false** if the *<key>* itself is not defined.

```
\keys_show:nn \keys_show:nn {<module>} {<key>}
```

Displays in the terminal the information associated to the *<key>* for a *<module>*, including the function which is used to actually implement it.

```
\keys_log:nn \keys_log:nn {<module>} {<key>}
```

Writes in the log file the information associated to the *<key>* for a *<module>*. See also `\keys_show:nn` which displays the result in the terminal.

28.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *<key–value list>* into *<keys>* and associated *<values>*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double `#` tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

key = {value here},

and

key = value here,

are treated identically.

<code>\keyval_parse:nnn</code>	<code>☆ \keyval_parse:nnn {⟨code₁⟩} {⟨code₂⟩} {⟨key-value list⟩}</code>

<code>\keyval_parse:(nnV nnv)</code>	<code>☆</code>

New: 2020-12-19

Updated: 2021-05-10

Parses the `⟨key-value list⟩` into a series of `⟨keys⟩` and associated `⟨values⟩`, or keys alone (if no `⟨value⟩` was given). `⟨code1⟩` receives each `⟨key⟩` (with no `⟨value⟩`) as a trailing brace group, whereas `⟨code2⟩` is appended by two brace groups, the `⟨key⟩` and `⟨value⟩`. The order of the `⟨keys⟩` in the `⟨key-value list⟩` is preserved. Thus

```
\keyval_parse:nnn
{ \use_none:nn { code 1 } }
{ \use_none:nnn { code 2 } }
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the `⟨key⟩` and `⟨value⟩`, then one *outer* set of braces is removed from the `⟨key⟩` and `⟨value⟩` as part of the processing. If you need exactly the output shown above, you'll need to either `e`-type or `x`-type expand the function.

TeXhackers note: The result of each list element is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

<code>\keyval_parse:NNn</code>	☆	<code>\keyval_parse:NNn <function₁> <function₂> {<key-value list>}</code>
<code>\keyval_parse:(NNV NNv)</code>	☆	Parses the <code><key-value list></code> into a series of <code><keys></code> and associated <code><values></code> , or keys alone (if no <code><value></code> was given). <code><function₁></code> should take one argument, while <code><function₂></code> should absorb two arguments. After <code>\keyval_parse:NNn</code> has parsed the <code><key-value list></code> , <code><function₁></code> is used to process keys given with no value and <code><function₂></code> is used to process keys given with a value. The order of the <code><keys></code> in the <code><key-value list></code> is preserved. Thus

Updated: 2021-05-10

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the `<key>` and `<value>`, then one *outer* set of braces is removed from the `<key>` and `<value>` as part of the processing.

This shares the implementation of `\keyval_parse:nnn`, the difference is only semantically.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an **e**-type or **x**-type argument expansion.

<code>\keyval_map_inline:nnn</code>	<code>\keyval_map_inline:nnn {<key-value list>} {<inline function₁>} {<inline function₂>}</code>

New: 2026-02-09

Applies the `<inline function1>` to every key in the `<key-value list>` which got no value, and `<inline function2>` to every key in the `<key-value list>` which got a value. The `<inline function1>` should consist of code that receives keys without a value as `#1`, and `<inline function2>` receives each key as `#1` and each value as `#2`.

<code>\keyval_map_break:</code> ☆	Used to terminate a <code>\keyval_map...</code> function before all entries in the <code><key-value list></code> have been processed. This normally takes place within a conditional statement, for example
New: 2026-02-09	

```

\keyval_map_inline:nnn { foo, bar = baz, bingo, boom, ... }
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}
{
  % Do more useful things
}

```

See also `\keyval_map_break:n`. Use outside of a `\keyval_map...` scenario leads to low level $\TeX$ errors.

$\TeX$ hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

<code>\keyval_map_break:n</code> ☆	<code>\keyval_map_break:n {<code>}</code>
New: 2026-02-09	Used to terminate a <code>\keyval_map...</code> function before all entries in the <code><key-value list></code> have been processed, inserting the <code><code></code> after the mapping has ended. This normally takes place within a conditional statement, for example

```

\keyval_map_inline:nnn { foo, bar = baz, bingo, boom, ... }
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \keyval_map_break:n { <code> } }
  {
    % Do something useful
  }
}
{
  % Do more useful things
}

```

Use outside of a `\keyval_map...` scenario leads to low level $\TeX$ errors.

$\TeX$ hackers note: When the mapping is broken, additional tokens may be inserted before the `<code>` is inserted into the input stream. This depends on the design of the mapping function.

Chapter 29

The `l3intarray` module

Fast global integer arrays

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast `l3seq`, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialization
- The absolute value of each entry has maximum $2^{30} - 1$ (i.e., one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

29.1 Creating and initializing integer array variables

<code>\intarray_new:Nn</code>	<code>\intarray_new:Nn</code>	<code>\intarray var</code>	<code>{\size}</code>

<code>\intarray_new:cn</code>

Evaluates the integer expression `\size` and allocates an *integer array variable* with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

<code>\intarray_const_from_clist:Nn</code>	<code>\intarray_const_from_clist:Nn</code>	<code>\intarray var</code>	<code>{\int expr clist}</code>

<code>\intarray_const_from_clist:cn</code>

Creates a new constant *integer array variable* or raises an error if the name is already taken. The *integer array variable* is set (globally) to contain as its items the results of evaluating each *integer expression* in the *comma list*.

<code>\intarray_gzero:N</code>	<code>\intarray_gzero:N</code>	<code>\intarray var</code>

<code>\intarray_gzero:c</code>

Sets all entries of the *integer array variable* to zero. Assignments are always global.

29.2 Adding data to integer arrays

<code>\intarray_gset:Nnn</code>	<code>\intarray_gset:Nnn <intarray var> {<position>} {<value>}</code>
<code>\intarray_gset:cnn</code>	Stores the result of evaluating the integer expression <code><value></code> into the <i><integer array variable></i> at the (integer expression) <code><position></code> . If the <code><position></code> is not between 1 and the <code>\intarray_count:N</code> , or the <code><value></code> 's absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

29.3 Counting entries in integer arrays

<code>\intarray_count:N *</code>	<code>\intarray_count:N <intarray var></code>
<code>\intarray_count:c *</code>	Expands to the number of entries in the <i><integer array variable></i> . Contrarily to <code>\seq_count:N</code> this is performed in constant time.

29.4 Using a single entry

<code>\intarray_item:Nn *</code>	<code>\intarray_item:Nn <intarray var> {<position>}</code>
<code>\intarray_item:cn *</code>	Expands to the integer entry stored at the (integer expression) <code><position></code> in the <i><integer array variable></i> . If the <code><position></code> is not between 1 and the <code>\intarray_count:N</code> , an error occurs.

<code>\intarray_rand_item:N *</code>	<code>\intarray_rand_item:N <intarray var></code>
<code>\intarray_rand_item:c *</code>	Selects a pseudo-random item of the <i><integer array></i> . If the <i><integer array></i> is empty, produce an error.

29.5 Integer array conditional

<code>\intarray_if_exist_p:N *</code>	<code>\intarray_if_exist_p:N <intarray var></code>
<code>\intarray_if_exist_p:c *</code>	<code>\intarray_if_exist:NTF <intarray var> {<true code>} {<false code>}</code>
<code>\intarray_if_exist:NTF *</code>	Tests whether the <i><intarray var></i> is currently defined. This does not check that the
<code>\intarray_if_exist:CTF *</code>	<i><intarray var></i> really is an integer array variable.

New: 2024-03-31

29.6 Viewing integer arrays

<code>\intarray_show:N</code>	<code>\intarray_show:N <intarray var></code>
<code>\intarray_show:c</code>	<code>\intarray_log:N <intarray var></code>
<code>\intarray_log:N</code>	Displays the items in the <i><integer array variable></i> in the terminal or writes them in
<code>\intarray_log:c</code>	the log file.

29.7 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `l3seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `l3intarray` module transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can “only” deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeX Live settings).

Chapter 30

The `l3fp` module

Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. *Floating point expressions* (“`<fp expr>`”) support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root $\sqrt{x}$, and parentheses.
 - Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
 - Boolean logic: sign `sign x`, negation `!x`, conjunction `x && y`, disjunction `x || y`, ternary operator `x ? y : z`.
 - Exponentials: `exp x`, `ln x`, x^y , `logb x`.
 - Integer factorial: `fact x`.
 - Trigonometry: `sin x`, `cos x`, `tan x`, `cot x`, `sec x`, `csc x` expecting their arguments in radians, and `sind x`, `cosd x`, `tand x`, `cotd x`, `secd x`, `cscd x` expecting their arguments in degrees.
 - Inverse trigonometric functions: `asin x`, `acos x`, `atan x`, `acot x`, `asec x`, `acsc x` giving a result in radians, and `asind x`, `acosd x`, `atand x`, `acotd x`, `asecd x`, `acscd x` giving a result in degrees.
- (*not yet*) Hyperbolic functions and their inverse functions: `sinh x`, `cosh x`, `tanh x`, `coth x`, `sech x`, `csch`, and `asinh x`, `acosh x`, `atanh x`, `acoth x`, `asech x`, `acsch x`.
- Extrema: `max(x1, x2, ...)`, `min(x1, x2, ...)`, `abs(x)`.
 - Rounding functions, controlled by two optional values, *n* (number of places, 0 by default) and *t* (behavior on a tie, `nan` by default):
 - `trunc(x, n)` rounds towards zero,
 - `floor(x, n)` rounds towards $-\infty$,

- `ceil( $x, n$ )` rounds towards $+\infty$,
- `round( $x, n, t$ )` rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And (*not yet*) modulo, and “quantize”.

- Random numbers: `rand()`, `randint( $m, n$ )`.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\langle type \rangle\_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: $(x_1, \dots, x_n)$ that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 30.13.1 for a description of what a floating point is, section 30.13.2 for details about how an expression is parsed, and section 30.13.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 30.11.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {\sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

30.1 Creating and initializing floating point variables

<u>\fp_new:N</u>	<u>\fp_new:N</u> $\langle fp\ var \rangle$
<u>\fp_new:c</u>	Creates a new $\langle fp\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle fp\ var \rangle$ is initially +0.
<u>\fp_const:Nn</u>	<u>\fp_const:Nn</u> $\langle fp\ var \rangle \{ \langle fp\ expr \rangle \}$
<u>\fp_const:cn</u>	Creates a new constant $\langle fp\ var \rangle$ or raises an error if the name is already taken. The $\langle fp\ var \rangle$ is set globally equal to the result of evaluating the $\langle fp\ expr \rangle$.
<u>\fp_zero:N</u>	<u>\fp_zero:N</u> $\langle fp\ var \rangle$
<u>\fp_zero:c</u>	Sets the $\langle fp\ var \rangle$ to +0.
<u>\fp_gzero:N</u>	
<u>\fp_gzero:c</u>	
<u>\fp_zero_new:N</u>	<u>\fp_zero_new:N</u> $\langle fp\ var \rangle$
<u>\fp_zero_new:c</u>	
<u>\fp_gzero_new:N</u>	Ensures that the $\langle fp\ var \rangle$ exists globally by applying $\backslash fp\_new:N$ if necessary, then applies $\backslash fp\_zero:N$ to leave the $\langle fp\ var \rangle$ set to +0.
<u>\fp_gzero_new:c</u>	

30.2 Setting floating point variables

<u>\fp_set:Nn</u>	<u>\fp_set:Nn</u> $\langle fp\ var \rangle \{ \langle fp\ expr \rangle \}$
<u>\fp_set:(cn NV cV)</u>	Sets $\langle fp\ var \rangle$ equal to the result of computing the $\langle fp\ expr \rangle$.
<u>\fp_gset:Nn</u>	
<u>\fp_gset:(cn NV cV)</u>	
<u>\fp_set_eq:NN</u>	<u>\fp_set_eq:NN</u> $\langle fp\ var_1 \rangle \langle fp\ var_2 \rangle$
<u>\fp_set_eq:(cn Nc cc)</u>	Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.
<u>\fp_gset_eq:NN</u>	
<u>\fp_gset_eq:(cn Nc cc)</u>	
<u>\fp_add:Nn</u>	<u>\fp_add:Nn</u> $\langle fp\ var \rangle \{ \langle fp\ expr \rangle \}$
<u>\fp_add:cn</u>	
<u>\fp_gadd:Nn</u>	Adds the result of computing the $\langle fp\ expr \rangle$ to the $\langle fp\ var \rangle$. This also applies if
<u>\fp_gadd:cn</u>	$\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.
<u>\fp_sub:Nn</u>	<u>\fp_sub:Nn</u> $\langle fp\ var \rangle \{ \langle fp\ expr \rangle \}$
<u>\fp_sub:cn</u>	
<u>\fp_gsub:Nn</u>	Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$.
<u>\fp_gsub:cn</u>	This also applies if $\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.

30.3 Using floating points

`\fp_eval:n` * `\fp_eval:n {<fp expr>}`

Evaluates the `<fp expr>` and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_eval:n` and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to `\fp_to_decimal:n`.

`\fp_sign:n` * `\fp_sign:n {<fp expr>}`

Evaluates the `<fp expr>` and leaves its sign in the input stream using `\fp_eval:n {sign(<result>)}`: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 . If the operand is a tuple or is `nan`, then “invalid operation” occurs and the result is 0.

`\fp_to_decimal:N` * `\fp_to_decimal:N <fp var>`
`\fp_to_decimal:c` * `\fp_to_decimal:n {<fp expr>}`
`\fp_to_decimal:n` *

Evaluates the `<fp expr>` and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

`\fp_to_dim:N` * `\fp_to_dim:N <fp var>`
`\fp_to_dim:c` * `\fp_to_dim:n {<fp expr>}`
`\fp_to_dim:n` *

Evaluates the `<fp expr>` and expresses the result as a dimension (in `pt`) suitable for use in dimension expressions. The output is identical to `\fp_to_decimal:n`, with an additional trailing `pt` (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid $\text{\TeX}$ dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and `nan`, trigger an “invalid operation” exception.

`\fp_to_int:N` * `\fp_to_int:N <fp var>`
`\fp_to_int:c` * `\fp_to_int:n {<fp expr>}`
`\fp_to_int:n` *

Evaluates the `<fp expr>`, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid $\text{\TeX}$ integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and `nan`, trigger an “invalid operation” exception.

`\fp_to_scientific:N` * `\fp_to_scientific:N` $\langle fp\ var \rangle$
`\fp_to_scientific:c` * `\fp_to_scientific:n` $\{\langle fp\ expr \rangle\}$
`\fp_to_scientific:n` * Evaluates the $\langle fp\ expr \rangle$ and expresses the result in scientific notation:

$\langle optional\ - \rangle \langle digit \rangle . \langle 15\ digits \rangle e \langle optional\ sign \rangle \langle exponent \rangle$

The leading $\langle digit \rangle$ is non-zero except in the case of ± 0 . The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

`\fp_to_tl:N` * `\fp_to_tl:N` $\langle fp\ var \rangle$
`\fp_to_tl:c` * `\fp_to_tl:n` $\{\langle fp\ expr \rangle\}$
`\fp_to_tl:n` * Evaluates the $\langle fp\ expr \rangle$ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from `\fp_to_scientific:n`). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`). Negative numbers start with `-`. The special values ± 0 , $\pm\infty$ and `nan` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, are made up of letters. For a tuple, each item is converted using `\fp_to_tl:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

`\fp_use:N` * `\fp_use:N` $\langle fp\ var \rangle$
`\fp_use:c` * Inserts the value of the $\langle fp\ var \rangle$ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to `\fp_to_decimal:N`.

30.4 Formatting floating points

`\fp_format:nn` ★ `\fp_format:nn {<fp expr>} {<format specification>}`

New: 2025-06-09

Evaluates the `<fp expr>` and converts the result to a string according to the `<format specification>`. The `<style>` can be

- **e** for scientific notation, with one digit before and `<precision>` digits after the decimal separator, and an integer exponent, following **e**;
- **f** for a fixed point notation, with `<precision>` digits after the decimal separator and no exponent;
- **g** for a general format, which uses style **f** for numbers in the range $[10^{-4}, 10^{<precision>})$ and style **e** otherwise.

When there is no `<style>` specifier nor `<precision>` the number is displayed without rounding. Otherwise the `<precision>` defaults to 6. The details of the `<format specification>` are described in Section 19.1.

30.5 Floating point conditionals

`\fp_if_exist_p:N` ★ `\fp_if_exist_p:N <fp var>`

`\fp_if_exist_p:c` ★ `\fp_if_exist:NTF <fp var> {<true code>} {<false code>}`

`\fp_if_exist:NTF` ★

`\fp_if_exist:cTF` ★ Tests whether the `<fp var>` is currently defined. This does not check that the `<fp var>` really is a floating point variable.

`\fp_compare_p:nNn` ★ `\fp_compare_p:nNn {<fp expr1>} <relation> {<fp expr2>}`

`\fp_compare:nNnTF` ★ `\fp_compare:nNnTF {<fp expr1>} <relation> {<fp expr2>} {<true code>} {<false code>}`

Compares the `<fp expr1>` and the `<fp expr2>`, and returns **true** if the `<relation>` is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is **nan** or is a tuple, unless they are equal tuples. Note that a **nan** is distinct from any value, even another **nan**, hence $x = x$ is not true for a **nan**. To test if a value is **nan**, compare it to an arbitrary number with the “not ordered” relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan
```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no **nan**). At present any other comparison with tuples yields ? (not ordered). This is experimental.

This function is less flexible than `\fp_compare:NNTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

```

\fp_compare_p:n * \fp_compare_p:n
\fp_compare:nTF * {
    <fp expr1> <relation1>
    ...
    <fp exprN> <relationN>
    <fp exprN+1>
}
\fp_compare:nTF
{
    <fp expr1> <relation1>
    ...
    <fp exprN> <relationN>
    <fp exprN+1>
}
{<true code>} {<false code>}

```

Evaluates the $\langle fp\ exprs \rangle$ as described for $\backslash fp\_eval:n$ and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle fp\ expr_1 \rangle$ and $\langle fp\ expr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle fp\ expr_2 \rangle$ and $\langle fp\ expr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle fp\ expr_N \rangle$ and $\langle fp\ expr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to $\backslash int\_compare:nTF$, all $\langle fp\ exprs \rangle$ are computed, even if one comparison is **false**. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is **nan** or is a tuple, unless they are equal tuples. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading $!$ (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floating point expressions. The comparison $x\ \langle relation \rangle\ y$ is then **true** if the $\langle relation \rangle$ does not start with $!$ and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with $!$ and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include $\geq$ (greater or equal), $\neq$ (not equal), $!? or \leq \Rightarrow$ (comparable).

This function is more flexible than $\backslash fp\_compare:nNnTF$ and only slightly slower.

```

\fp_if_nan_p:n * \fp_if_nan_p:n {<fp expr>}
\fp_if_nan:nTF * \fp_if_nan:nTF {<fp expr>} {<true code>} {<false code>}

```

Evaluates the $\langle fp\ expr \rangle$ and tests whether the result is exactly **nan**. The test returns **false** for any other result, even a tuple containing **nan**.

30.6 Floating point expression loops

```

\fp_do_until:nNnn ☆ \fp_do_until:nNnn {<fp expr1>} <relation> {<fp expr2>} {<code>}

```

Places the $\langle code \rangle$ in the input stream for $\text{\TeX}$ to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for $\backslash fp\_compare:nNnTF$. If the test is **false** then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is **true**.

<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\fp_until_do:nNnn</code> ☆	<code>\fp_until_do:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\fp_while_do:nNnn</code> ☆	<code>\fp_while_do:nNnn {<fp expr₁>} <relation> {<fp expr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is <code>false</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>true</code> .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
	Places the <code><code></code> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is <code>true</code> then the <code><code></code> is inserted into the input stream again and a loop occurs until the <code><relation></code> is <code>false</code> .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>false</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\fp_while_do:nn</code> ☆	<code>\fp_while_do:nn { <fp expr₁> <relation> <fp expr₂> } {<code>}</code>
	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <code><code></code> in the input stream if the <code><relation></code> is <code>true</code> . After the <code><code></code> has been processed by T _E X the test is repeated, and a loop occurs until the test is <code>false</code> .

<code>\fp_step_function:nnnN</code> ☆ <code>\fp_step_function:nnnc</code> ☆	<code>\fp_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩</code>

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0]   [I saw 1.1]   [I saw 1.2]   [I saw 1.3]   [I saw 1.4]   [I saw 1.5]
```

TpXhackers note: Due to rounding, it may happen that adding the $\langle step \rangle$ to the $\langle value \rangle$ does not change the $\langle value \rangle$; such cases give an error, as they would otherwise lead to an infinite loop.

<code>\fp_step_inline:nnnn</code>	<code>\fp_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with $\#1$ replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument ($\#1$).

<code>\fp_step_variable:nnnNn</code>	<code>\fp_step_variable:nnnNn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

30.7 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard $\backslash l_...$ variables.

```
\fp_new_variable:n { A }
\fp_set:Nn \l_tmpb_fp { 1 * sin(A) + 3**2 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { pi/2 }
```

```

\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { 0 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp

```

defines `A` to be a variable, then defines `\l_tmpb_fp` to stand for $1 \cdot \sin(A) + 9$ (note that $3 \cdot 2$ is evaluated, but the $1 \cdot$ product is not simplified away). Until `\l_tmpb_fp` is changed, `\fp_show:N \l_tmpb_fp` will show $((1 \cdot \sin(A)) + 9)$ regardless of the value of `A`. The next step defines `A` to be equal to $\pi/2$: then `\fp_show:n { \l_tmpb_fp }` will evaluate `\l_tmpb_fp` and show 10. We then redefine `A` to be 0: since `\l_tmpb_fp` still stands for $1 \cdot \sin(A) + 9$, the value shown is then 9. Variables can be set with `\fp_set_variable:nn` to arbitrary floating point expressions including other variables.

At present, the following operations and functions are *not* supported

- infix binary comparisons like `>`, `=`, `<`
- infix ternary operator like `?:`
- prefix variable-ary functions like `round` and friends, `min`, `max`, `atan`, `acot`, `atand`, `acotd`

```
\fp_new_variable:n \fp_new_variable:n {<identifier>}
```

New: 2023-10-19

Declares the `<identifier>` as a variable, which allows it to be used in floating point expressions. For instance,

```

\fp_new_variable:n { A }
\fp_show:n { A**2 - A + 1 }

```

shows $((A^2) - A) + 1$. If the declaration was missing, the parser would complain about an “Unknown fp word ‘A’”. The `<identifier>` must consist entirely of Latin letters among `[a-zA-Z]`.

```
\fp_set_variable:nn \fp_set_variable:nn {<identifier>} {<fp expr>}
```

New: 2023-10-19

Sets the `<identifier>` to stand in any further expression for the result of evaluating the `<floating point expression>` as much as possible. The `<identifier>` must be declared by `\fp_new_function:n` first. The result may contain other variables, which are then replaced by their values if they have any. For instance,

```

\fp_new_variable:n { A }
\fp_new_variable:n { B }
\fp_new_variable:n { C }
\fp_set_variable:nn { A } { 3 }
\fp_set_variable:nn { C } { A ** 2 + B * 1 }
\fp_show:n { C + 4 }
\fp_set_variable:nn { A } { 4 }
\fp_show:n { C + 4 }

```

shows $((9 + (B \cdot 1)) + 4)$ twice: changing the value of `A` to 4 does not alter `C` because `A` was replaced by its value 3 when evaluating $A \cdot 2 + B \cdot 1$.

<code>\fp_clear_variable:n</code>	<code>\fp_clear_variable:n {⟨<i>identifier</i>⟩}</code>
New: 2023-10-19	Removes any value given by <code>\fp_set_variable:nn</code> to the variable with this <i>⟨identifier⟩</i> . For instance,
	<pre> \fp_new_variable:n { A } \fp_set_variable:nn { A } { 3 } \fp_show:n { A ^ 2 } \fp_clear_variable:n { A } \fp_show:n { A ^ 2 } </pre>

shows 9, then (A^2) .

30.8 User-defined functions

It is possible to define new user functions which can be used inside the argument to `\fp_eval:n`, etc. These functions may take one or more named arguments, and should be implemented using expansion methods only.

<code>\fp_new_function:n</code>	<code>\fp_new_function:n {⟨<i>identifier</i>⟩}</code>
New: 2023-10-19	Declares the <i>⟨identifier⟩</i> as a function, which allows it to be used in floating point expressions. For instance,
	<pre> \fp_new_function:n { foo } \fp_show:n { foo (1 + 2 , foo(3), A) ** 2 } } </pre>
	shows $(\text{foo}(3, \text{foo}(3), A))^2$. If the declaration was missing, the parser would complain about an “Unknown fp word ‘foo’”. The <i>⟨identifier⟩</i> must consist entirely of Latin letters [a-zA-Z].

<code>\fp_set_function:nnn</code>	<code>\fp_set_function:nnn {⟨<i>identifier</i>⟩} {⟨<i>vars</i>⟩} {⟨<i>fp expr</i>⟩}</code>
New: 2023-10-19	Sets the <i>⟨identifier⟩</i> to stand in any further expression for the result of evaluating the <i>⟨floating point expression⟩</i> , with the <i>⟨identifier⟩</i> accepting the <i>⟨vars⟩</i> (a non-empty comma-separated list). The <i>⟨identifier⟩</i> must be declared by <code>\fp_new_function:n</code> first. The result may contain other functions, which are then replaced by their results if they have any. For instance,
	<pre> \fp_new_function:n { npow } \fp_set_function:nnn { npow } { a,b } { a**b } \fp_show:n { npow(16,0.25) } </pre>
	shows 2. The names of the <i>⟨vars⟩</i> must consist entirely of Latin letters [a-zA-Z], but are otherwise not restricted: in particular, they are independent of any variables declared by <code>\fp_new_variable:n</code> .

<code>\fp_clear_function:n</code>	<code>\fp_clear_function:n {⟨<i>identifier</i>⟩}</code>
New: 2023-10-19	Removes any definition given by <code>\fp_set_function:nnn</code> to the function with this <i>⟨identifier⟩</i> .

30.9 Some useful constants, and scratch variables

<code>\c_zero_fp</code>	Zero, with either sign.
<code>\c_minus_zero_fp</code>	

<code>\c_one_fp</code>	One as an <code>fp</code> : useful for comparisons in some places.

<code>\c_inf_fp</code>	Infinity, with either sign. These can be input directly in a floating point expression as
<code>\c_minus_inf_fp</code>	<code>inf</code> and <code>-inf</code> .

<code>\c_nan_fp</code>	Not a number. This can be input directly in a floating point expression as <code>nan</code> .

<code>\c_e_fp</code>	The value of the base of the natural logarithm, $e = \exp(1)$.

<code>\c_pi_fp</code>	The value of π . This can be input directly in a floating point expression as <code>pi</code> .

<code>\c_one_degree_fp</code>	The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as <code>deg</code> .

30.10 Scratch variables

<code>\l_tmpa_fp</code>	Scratch floating points for local assignment. These are never used by the kernel code, and
<code>\l_tmpb_fp</code>	so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

<code>\g_tmpa_fp</code>	Scratch floating points for global assignment. These are never used by the kernel code,
<code>\g_tmpb_fp</code>	and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

30.11 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or `10 ** 1e9999`. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, e.g., $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `\l_fp_overflow_flag`, `\l_fp_underflow_flag`, `\l_fp_invalid_operation_flag` and `\l_fp_division_by_zero_flag`. The state of these flags can be tested and modified with commands from `l3flag`

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behavior when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:nn` `\fp_trap:nn` $\langle exception \rangle$ $\langle trap\ type \rangle$

All occurrences of the $\langle exception \rangle$ (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as $\langle trap\ type \rangle$, which can be

- **none**: the $\langle exception \rangle$ will be entirely ignored, and leave no trace;
- **flag**: the $\langle exception \rangle$ will turn the corresponding flag on when it occurs;
- **error**: additionally, the $\langle exception \rangle$ will halt the T_EX run and display some information about the current operation in the terminal.

`\l_fp_overflow_flag`
`\l_fp_underflow_flag`
`\l_fp_invalid_operation_flag`
`\l_fp_division_by_zero_flag`

Flags denoting the occurrence of various floating-point exceptions.

30.12 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N <fp var></code>
<code>\fp_show:c</code>	<code>\fp_show:n {<fp expr>}</code>
<code>\fp_show:n</code>	Evaluates the <code><fp expr></code> and displays the result in the terminal.

Updated: 2021-04-29

<code>\fp_log:N</code>	<code>\fp_log:N <fp var></code>
<code>\fp_log:c</code>	<code>\fp_log:n {<fp expr>}</code>
<code>\fp_log:n</code>	Evaluates the <code><fp expr></code> and writes the result in the log file.

Updated: 2021-04-29

30.13 Floating point expressions

30.13.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- **nan**, is “not a number”, and can be either quiet or signaling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- `<sign>`: a possibly empty string of + and - characters;
- `<significand>`: a non-empty string of digits together with zero or one dot;
- `<exponent>` optionally: the character **e** or **E**, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if `<sign>` contains an even number of -, and - otherwise, hence, an empty `<sign>` denotes a non-negative input. The stored significand is obtained from `<significand>` by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input `<significand>` has at most 16 digits. The stored `<exponent>` is obtained by combining the input `<exponent>` (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting `<exponent>` is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if `<significand>` contains no non-zero digit (i.e., consists only in characters 0, and an optional period), or if there is an underflow. Note

that a single dot is currently a valid floating point number, equal to $+0$, but that is not guaranteed to remain true.

The `<significand>` must be non-empty, so `e1` and `e-1` are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as `e` and should be input as `exp(1)` or `\c_e_fp` (which is faster).

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any `<sign>`, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `nan`.
- Note that commands such as `\infty`, `\pi`, or `\sin` *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

30.13.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned} 1/2\text{pi} &= 1/(2\pi), \\ 1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \sin 2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^2\text{max}(3, 5) &= 2^2 \max(3, 5) = 20, \\ 1\text{in}/1\text{cm} &= (1\text{in})/(1\text{cm}) = 2.54. \end{aligned}$$

Functions are called on the value of their argument, contrarily to $\text{\TeX}$ macros.

30.13.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is **false** if it is ± 0 , and **true** otherwise, including when it is **nan** or a tuple such as $(0, 0)$. Tuples are only supported to some extent by operations that work with truth values ($?:$, $||$, $\&\&$, $!$), by comparisons ($!<=>?$), and by $+$, $-$, $*$, $/$. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a **nan** result.

```
?: \fp_eval:n { <operand_1> ? <operand_2> : <operand_3> }
```

The ternary operator $?:$ results in $\langle \text{operand}_2 \rangle$ if $\langle \text{operand}_1 \rangle$ is true (not ± 0), and $\langle \text{operand}_3 \rangle$ if $\langle \text{operand}_1 \rangle$ is false (± 0). All three $\langle \text{operands} \rangle$ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following $:$ is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before $:$ is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
|| \fp_eval:n { <operand_1> || <operand_2> }
```

If $\langle \text{operand}_1 \rangle$ is true (not ± 0), use that value, otherwise the value of $\langle \text{operand}_2 \rangle$. Both $\langle \text{operands} \rangle$ are evaluated in all cases; they may be tuples. In $\langle \text{operand}_1 \rangle || \langle \text{operand}_2 \rangle || \dots || \langle \text{operands}_n \rangle$, the first true (nonzero) $\langle \text{operand} \rangle$ is used and if all are zero the last one (± 0) is used.

```
&& \fp_eval:n { <operand_1> && <operand_2> }
```

If $\langle \text{operand}_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle \text{operand}_2 \rangle$. Both $\langle \text{operands} \rangle$ are evaluated in all cases; they may be tuples. In $\langle \text{operand}_1 \rangle \&\& \langle \text{operand}_2 \rangle \&\& \dots \&\& \langle \text{operands}_n \rangle$, the first false (± 0) $\langle \text{operand} \rangle$ is used and if none is zero the last one is used.

```

< \fp_eval:n
= {
>   <operand1> <relation1>
?   ...
-   <operandN> <relationN>
    <operandN+1>
}

```

Each $\langle \text{relation} \rangle$ consists of a non-empty string of $<$, $=$, $>$, and $?$, optionally preceded by $!$, and may not start with $?$. This evaluates to $+1$ if all comparisons $\langle \text{operand}_i \rangle \langle \text{relation}_i \rangle \langle \text{operand}_{i+1} \rangle$ are true, and $+0$ otherwise. All $\langle \text{operands} \rangle$ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```

+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }

```

Computes the sum or the difference of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is **nan**.

```

* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }

```

Computes the product or the ratio of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When $\langle \text{operand}_1 \rangle$ is a tuple and $\langle \text{operand}_2 \rangle$ is a floating point number, each item of $\langle \text{operand}_1 \rangle$ is multiplied or divided by $\langle \text{operand}_2 \rangle$. Multiplication also supports the case where $\langle \text{operand}_1 \rangle$ is a floating point number and $\langle \text{operand}_2 \rangle$ a tuple. Other combinations yield an “invalid operation” exception and a **nan** result.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle \text{operand} \rangle$ (for a tuple, of all its components), and $!$ $\langle \text{operand} \rangle$ evaluates to 1 if $\langle \text{operand} \rangle$ is false (is ± 0) and 0 otherwise (this is the **not** boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle \text{operand}_1 \rangle$ to the power $\langle \text{operand}_2 \rangle$. This operation is right associative, hence $2^{**} 2^{**} 3$ equals $2^{2^3} = 256$. If $\langle \text{operand}_1 \rangle$ is negative or -0 then: the result’s sign is $+$ if the $\langle \text{operand}_2 \rangle$ is infinite and $(-1)^p$ if the $\langle \text{operand}_2 \rangle$ is $p/5^q$ with p, q integers; the result is $+0$ if $\text{abs}(\langle \text{operand}_1 \rangle)^{\langle \text{operand}_2 \rangle}$ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

```

abs \fp_eval:n { abs( <fp expr> ) }

```

Computes the absolute value of the $\langle \text{fp expr} \rangle$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

	<code>\fp_eval:n { exp(<fp expr>) }</code>	
		Computes the exponential of the $\langle fp \text{ expr} \rangle$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
	<code>\fp_eval:n { fact(<fp expr>) }</code>	
		Computes the factorial of the $\langle fp \text{ expr} \rangle$. If the $\langle fp \text{ expr} \rangle$ is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with “overflow”, while $\text{fact}(+\infty) = +\infty$ and $\text{fact}(\text{nan}) = \text{nan}$ with no exception. All other inputs give nan with the “invalid operation” exception.
	<code>\fp_eval:n { ln(<fp expr>) }</code>	
		Computes the natural logarithm of the $\langle fp \text{ expr} \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
	<code>\logb * \fp_eval:n { logb(<fp expr>) }</code>	
		Determines the exponent of the $\langle fp \text{ expr} \rangle$, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating $\logb(\pm 0) = -\infty$. Other special values are $\logb(\pm\infty) = +\infty$ and $\logb(\text{nan}) = \text{nan}$. If the operand is a tuple or is nan , then “invalid operation” occurs and the result is nan .
	<code>\fp_eval:n { max(<fp expr₁> , <fp expr₂> , ...) }</code>	
	<code>\fp_eval:n { min(<fp expr₁> , <fp expr₂> , ...) }</code>	
		Evaluates each $\langle fp \text{ expr} \rangle$ and computes the largest (smallest) of those. If any of the $\langle fp \text{ expr} \rangle$ is a nan or tuple, the result is nan . If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

```

round \fp_eval:n { round ( <fp expr> ) }
trunc \fp_eval:n { round ( <fp expr_1> , <fp expr_2> ) }
ceil  \fp_eval:n { round ( <fp expr_1> , <fp expr_2> , <fp expr_3> ) }
floor

```

Only `round` accepts a third argument. Evaluates $\langle fp\ expr_1 \rangle = x$ and $\langle fp\ expr_2 \rangle = n$ and $\langle fp\ expr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or `nan`; if $n = \text{nan}$, this yields `nan`; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fp\ expr_2 \rangle$ is omitted, $n = 0$, i.e., $\langle fp\ expr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- `round` yields the multiple of 10^{-n} closest to x , with ties (x half-way between two such multiples) rounded as follows. If t is `nan` (or not given) the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”).
- `floor` yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- `ceil` yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- `trunc` yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).

“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fp\ expr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.

```

sign \fp_eval:n { sign( <fp expr> ) }

```

Evaluates the $\langle fp\ expr \rangle$ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and `nan` for `nan`. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

```

sin \fp_eval:n { sin( <fp expr> ) }
cos \fp_eval:n { cos( <fp expr> ) }
tan \fp_eval:n { tan( <fp expr> ) }
cot \fp_eval:n { cot( <fp expr> ) }
csc \fp_eval:n { csc( <fp expr> ) }
sec \fp_eval:n { sec( <fp expr> ) }

```

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fp\ expr \rangle$ given in radians. For arguments given in degrees, see `sind`, `cosd`, etc. Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analogue `sind( $8 \times 180$ )` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

sind \fp_eval:n { sind( <fp expr> ) }
cosd \fp_eval:n { cosd( <fp expr> ) }
tand \fp_eval:n { tand( <fp expr> ) }
cotd \fp_eval:n { cotd( <fp expr> ) }
cscd \fp_eval:n { cscd( <fp expr> ) }
secd \fp_eval:n { secd( <fp expr> ) }

```

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fp\ expr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, etc. Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8 × 180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

asin \fp_eval:n { asin( <fp expr> ) }
acos \fp_eval:n { acos( <fp expr> ) }
acsc \fp_eval:n { acsc( <fp expr> ) }
asec \fp_eval:n { asec( <fp expr> ) }

```

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fp\ expr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, etc. If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

asind \fp_eval:n { asind( <fp expr> ) }
acosd \fp_eval:n { acosd( <fp expr> ) }
acscd \fp_eval:n { acscd( <fp expr> ) }
asecd \fp_eval:n { asecd( <fp expr> ) }

```

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fp\ expr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asind` and `acscd` and $[0, 180]$ for `acosd` and `asecd`. For a result in radians, use `asin`, etc. If the argument of `asind` or `acosd` lies outside the range $[-1, 1]$, or the argument of `acscd` or `asecd` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

atan \fp_eval:n { atan( <fp expr> ) }
acot \fp_eval:n { atan( <fp expr1> , <fp expr2> ) }


---


\fp_eval:n { acot( <fp expr> ) }
\fp_eval:n { acot( <fp expr1> , <fp expr2> ) }

```

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the `<fp expr>`: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fp \ expr_2 \rangle, \langle fp \ expr_1 \rangle)$: this is the arctangent of $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fp \ expr_1 \rangle$ and $\langle fp \ expr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fp \ expr_1 \rangle, \langle fp \ expr_2 \rangle)$, equal to the arccotangent of $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

```

atand \fp_eval:n { atand( <fp expr> ) }
acotd \fp_eval:n { atand( <fp expr1> , <fp expr2> ) }


---


\fp_eval:n { acotd( <fp expr> ) }
\fp_eval:n { acotd( <fp expr1> , <fp expr2> ) }

```

Those functions yield an angle in degrees: `atan` and `acot` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the `<fp expr>`: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fp \ expr_2 \rangle, \langle fp \ expr_1 \rangle)$: this is the arctangent of $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fp \ expr_1 \rangle$ and $\langle fp \ expr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fp \ expr_1 \rangle, \langle fp \ expr_2 \rangle)$, equal to the arccotangent of $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fp \ expr_1 \rangle / \langle fp \ expr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

```

sqrt \fp_eval:n { sqrt( <fp expr> ) }


---



```

Computes the square root of the `<fp expr>`. The “invalid operation” is raised when the `<fp expr>` is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{nan}} = \text{nan}$.

rand `\fp_eval:n { rand() }`

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of $\text{X}\text{\_}\text{T}\text{E}\text{X}$. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

T_EXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive `\pdfuniformdeviate` in $\text{pdf}\text{T}\text{E}\text{X}$, $\text{p}\text{T}\text{E}\text{X}$, $\text{up}\text{T}\text{E}\text{X}$ and `\uniformdeviate` in $\text{Lua}\text{T}\text{E}\text{X}$ and $\text{X}\text{\_}\text{T}\text{E}\text{X}$. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint `\fp_eval:n { randint( <fp expr> ) }`
`\fp_eval:n { randint( <fp expr1> , <fp expr2> ) }`

Produces a pseudo-random integer between 1 and $\langle \text{fp expr} \rangle$ or between $\langle \text{fp expr}_1 \rangle$ and $\langle \text{fp expr}_2 \rangle$ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See **rand** for important comments on how these pseudo-random numbers are generated.

inf The special values $+\infty$, $-\infty$, and **nan** are represented as **inf**, **-inf** and **nan** (see `\c_-nan`
`inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

em Those units of measurement are equal to their values in **pt**, namely

ex

in $1\text{ in} = 72.27\text{ pt}$

pt $1\text{ pt} = 1\text{ pt}$

pc $1\text{ pc} = 12\text{ pt}$

cm

mm $1\text{ cm} = \frac{1}{2.54}\text{ in} = 28.45275590551181\text{ pt}$

dd

cc $1\text{ mm} = \frac{1}{25.4}\text{ in} = 2.845275590551181\text{ pt}$

nd

nc $1\text{ dd} = 0.376065\text{ mm} = 1.07000856496063\text{ pt}$

bp $1\text{ cc} = 12\text{ dd} = 12.84010277952756\text{ pt}$

sp $1\text{ nd} = 0.375\text{ mm} = 1.066978346456693\text{ pt}$

$1\text{ nc} = 12\text{ nd} = 12.80374015748031\text{ pt}$

$1\text{ bp} = \frac{1}{72}\text{ in} = 1.00375\text{ pt}$

$1\text{ sp} = 2^{-16}\text{ pt} = 1.52587890625 \times 10^{-5}\text{ pt}.$

The values of the (font-dependent) units **em** and **ex** are gathered from $\text{\TeX}$ when the surrounding floating point expression is evaluated.

true Other names for 1 and +0.

false

\fp_abs:n $\star$ **\fp_abs:n** $\{\langle fp\ expr \rangle\}$

Evaluates the $\langle fp\ expr \rangle$ as described for **\fp_eval:n** and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, **nan** or a tuple, “invalid operation” occurs. Within floating point expressions, **abs()** can be used; it accepts $\pm\infty$ and **nan** as arguments.

\fp_max:nn $\star$ **\fp_max:nn** $\{\langle fp\ expr_1 \rangle\} \{\langle fp\ expr_2 \rangle\}$

\fp_min:nn $\star$ Evaluates the $\langle fp\ exprs \rangle$ as described for **\fp_eval:n** and leaves the resulting larger (**max**) or smaller (**min**) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, **max()** and **min()** can be used.

30.14 Disclaimer and roadmap

This module may break if the escape character is among **0123456789_+**, or if it receives a $\text{\TeX}$ primitive conditional affected by **\exp_not:N**.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signaling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {<fp expr>} {<format>}`, but what should `<format>` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add `log(x,b)` for logarithm of x in base b .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {Opt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a $\text{T}_{\text{E}}\text{X}$ “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection [30.13.1](#), write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/([200x] + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `\_fp\_basics\_pack\_weird\_low:NNNNw` and `\_fp\_basics\_pack\_weird\_high:NNNNNNNNw` better. Move the other `basics\_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan’s articles, some previous TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Chapter 31

The `l3fparray` module

Fast global floating point arrays

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast `l3seq`, where access time is linear). The interface is very close to that of `l3intarray`. The size of the array is fixed and must be given at point of initialization

31.1 Creating and initializing floating point array variables

<code>\fparray_new:Nn</code>	<code>\fparray_new:Nn <fparray var> {<size>}</code>
<code>\fparray_new:cn</code>	Evaluates the integer expression <code><size></code> and allocates an <code><fparray var></code> with that number of (zero) entries. The variable name should start with <code>\g_</code> because assignments are always global.
<code>\fparray_gzero:N</code>	<code>\fparray_gzero:N <fparray var></code>
<code>\fparray_gzero:c</code>	Sets all entries of the <code><fparray var></code> to <code>+0</code> . Assignments are always global.

31.2 Adding data to floating point arrays

<code>\fparray_gset:Nnn</code>	<code>\fparray_gset:Nnn <fparray var> {<position>} {<value>}</code>
<code>\fparray_gset:cnn</code>	Stores the result of evaluating the floating point expression <code><value></code> into the <code><fparray var></code> at the (integer expression) <code><position></code> . If the <code><position></code> is not between 1 and the <code>\fparray_count:N</code> , an error occurs. Assignments are always global.

31.3 Counting entries in floating point arrays

<code>\fpararray_count:N</code>	★	<code>\fpararray_count:N</code> $\langle fpararray\ var \rangle$
<code>\fpararray_count:c</code>	★	Expands to the number of entries in the $\langle fpararray\ var \rangle$. This is performed in constant time.

31.4 Using a single entry

<code>\fpararray_item:Nn</code>	★	<code>\fpararray_item:Nn</code> $\langle fpararray\ var \rangle$ $\{\langle position \rangle\}$
<code>\fpararray_item:cn</code>	★	Applies <code>\fp_use:N</code> or <code>\fp_to_tl:N</code> (respectively) to the floating point entry stored at the (integer expression) $\langle position \rangle$ in the $\langle fpararray\ var \rangle$. If the $\langle position \rangle$ is not between 1 and the <code>\fpararray_count:N</code> $\langle fpararray\ var \rangle$, an error occurs.
<code>\fpararray_item_to_tl:Nn</code>	★	
<code>\fpararray_item_to_tl:cn</code>	★	

31.5 Floating point array conditional

<code>\fpararray_if_exist_p:N</code>	★	<code>\fpararray_if_exist_p:N</code> $\langle fpararray\ var \rangle$
<code>\fpararray_if_exist_p:c</code>	★	<code>\fpararray_if_exist:NTF</code> $\langle fpararray\ var \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\fpararray_if_exist:NTF</code>	★	Tests whether the $\langle fpararray\ var \rangle$ is currently defined. This does not check that the
<code>\fpararray_if_exist:cTF</code>	★	$\langle fpararray\ var \rangle$ really is a floating point array variable.

New: 2024-03-31

Chapter 32

The **l3bitset** module

Bitsets

This module defines and implements the data type **bitset**, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names **1**, **2**, **3**, ... are predeclared and point to the index positions 1, 2, 3, ... More names can be added and existing names can be changed. The index is like all other indices in **expl3** modules *1-based*. A **bitset** can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the **bitset** package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The **bitset** is stored as a string (but one shouldn't rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by **TeX**-memory. But the functions to set and clear bits use integer functions for the index so bitsets can't be longer than $2^{31} - 1$. The export function **\bitset_to_arabic:N** can use functions from the **int** module only if the largest index used for this **bitset** is smaller than 32, for longer bitsets **fp** is used and this is slower.

32.1 Creating bitsets

```

\bitset_new:N \bitset_new:N <bitset var>
\bitset_new:c \bitset_new:Nn <bitset var>
\bitset_new:Nn {
\bitset_new:cn   <name1> = <index1> ,
                  <name2> = <index2> , ...
New: 2023-11-15 }

```

Creates a new *<bitset var>* or raises an error if the name is already taken. The declaration is global. The *<bitset var>* is initially 0.

Bitsets are implemented as string variables consisting of 1's and 0's. The rightmost number is the index position 1, so the string variable can be viewed directly as the binary number. But one shouldn't rely on the internal representation, but use the dedicated *\bitset_to_bin:N* instead to get the binary number.

The name-index pairs given in the second argument of *\bitset_new:Nn* declares names for some indices, which can be used to set and unset bits. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3, ...

<index...> should be a positive number or an *<integer expression>* which evaluates to a positive number. The expression is evaluated when the index is used, not at declaration time. The names *<name...>* should be unique. Using a number as name, e.g. 10=1, is allowed, it then overwrites the predeclared name 10, but the index position 10 can then only be reached if some other name for it exists, e.g. `ten=10`. It is not necessary to give every index a name, and an index can have more than one name. The named index can be extended or changed with the next function.

```

\bitset_addto_named_index:Nn \bitset_addto_named_index:Nn <bitset var>
New: 2023-11-15 {
                  <name1> = <index1> ,
                  <name2> = <index2> , ...
}

```

This extends or changes the name-index pairs for *<bitset var>* globally as described for *\bitset_new:Nn*.

For example after these settings

```

\bitset_new:Nn \l_pdfannot_F_bitset
{
  Invisible      = 1,
  Hidden        = 2,
  Print          = 3,
  NoZoom         = 4,
  NoRotate       = 5,
  NoView         = 6,
  ReadOnly       = 7,
  Locked         = 8,
  ToggleNoView   = 9,
  LockedContents = 10
}
\bitset_addto_named_index:Nn \l_pdfannot_F_bitset
{

```

```

    print = 3
}

```

it is possible to set bit 3 by using any of these alternatives:

```

\bitset_set_true:Nn \l_pdfannot_F_bitset {Print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {3}

```

```

\bitset_if_exist_p:N * \bitset_if_exist_p:N <bitset var>
\bitset_if_exist_p:c * \bitset_if_exist:NTF <bitset var> {<true code>} {<false code>}
\bitset_if_exist:NTF * Tests whether the <bitset var> exist.
\bitset_if_exist:cTF *

```

New: 2023-11-15

32.2 Setting and unsetting bits

```

\bitset_set_true:Nn \bitset_set_true:Nn <bitset var> {<name>}
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn

```

This sets the bit of the index position represented by {<name>} to 1. <name> should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. Index position are 1-based. If needed the length of the bit vector is enlarged.

New: 2023-11-15

```

\bitset_set_false:Nn \bitset_set_false:Nn <bitset var> {<name>}
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn

```

This unsets the bit of the index position represented by {<name>} (sets it to 0). <name> should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. The index is 1-based. If the index position is larger than the current length of the bit vector nothing happens. If the leading (left most) bit is unset, zeros are not trimmed but stay in the bit vector and are still shown by \bitset_show:N.

New: 2023-11-15

```

\bitset_clear:N \bitset_clear:N <bitset var>
\bitset_clear:c
\bitset_gclear:N
\bitset_gclear:c

```

This resets the bitset to the initial state. The declared names are not changed.

New: 2023-11-15

32.3 Using bitsets

```

\bitset_item:Nn * \bitset_item:Nn <bitset var> {<name>}
\bitset_item:cn *

```

\bitset_item:Nn outputs 1 if the bit with the index number represented by <name> is set and 0 otherwise. <name> is either one of the predeclared names 1, 2, 3, ..., or one of the names added manually.

New: 2023-11-15

<code>\bitset_to_bin:N</code>	★	<code>\bitset_to_bin:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_to_bin:c</code>	★	This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero.

New: 2023-11-15

<code>\bitset_to_arabic:N</code>	★	<code>\bitset_to_arabic:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_to_arabic:c</code>	★	This leaves the current value of the bitset expressed as a decimal number in the input stream. If no bit has been set yet, the output is zero. The function uses <code>\int_from_bin:n</code> if the largest index that have been set or unset is smaller than 32, and a slower implementation based on <code>\fp_eval:n</code> otherwise.

New: 2023-11-15

<code>\bitset_use:N</code>	★	<code>\bitset_use:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_use:c</code>	★	This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero. This is functionally equivalent to <code>\bitset_to_bin:N</code> .

New: 2024-11-12

<code>\bitset_show:N</code>		<code>\bitset_show:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_show:c</code>		Displays the binary and decimal values of the <i>⟨bitset var⟩</i> on the terminal.

New: 2023-11-15

<code>\bitset_log:N</code>		<code>\bitset_log:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_log:c</code>		Writes the binary and decimal values of the <i>⟨bitset var⟩</i> in the log file.

New: 2023-11-15

<code>\bitset_show_named_index:N</code>		<code>\bitset_show_named_index:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_show_named_index:c</code>		Displays declared name–index pairs of the <i>⟨bitset var⟩</i> on the terminal.

New: 2023-11-15

<code>\bitset_log_named_index:N</code>		<code>\bitset_log_named_index:N</code> <i>⟨bitset var⟩</i>
<code>\bitset_log_named_index:c</code>		Writes declared name–index pairs of the <i>⟨bitset var⟩</i> in the log file.

New: 2023-12-11

Chapter 33

The `\l3cctab` module

Category code tables

A category code table enables rapid switching of all category codes in one operation. For Lua_T_E_X, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in `expl3` also saves and restores the `TEX \endlinechar` primitive value, meaning they could be used for example to implement `\ExplSyntaxOn`.

33.1 Creating and initializing category code tables

<code>\cctab_new:N</code>	<code>\cctab_new:N <category code table></code>
<code>\cctab_new:c</code>	Creates a new <code><category code table></code> variable or raises an error if the name is already taken. The declaration is global. The <code><category code table></code> is initialized with the codes as used by <code>iniT_EX</code> .
<code>Updated: 2020-07-02</code>	

<code>\cctab_const:Nn</code>	<code>\cctab_const:Nn <category code table> {<category code set up>}</code>
<code>\cctab_const:cn</code>	Creates a new <code><category code table></code> , applies (in a group) the <code><category code set up></code> on top of <code>iniT_EX</code> settings, then saves them globally as a constant table. The <code><category code set up></code> can include a call to <code>\cctab_select:N</code> .
<code>Updated: 2020-07-07</code>	

<code>\cctab_gset:Nn</code>	<code>\cctab_gset:Nn <category code table> {<category code set up>}</code>
<code>\cctab_gset:cn</code>	Starting from the <code>iniT_EX</code> category codes, applies (in a group) the <code><category code set up></code> , then saves them globally in the <code><category code table></code> . The <code><category code set up></code> can include a call to <code>\cctab_select:N</code> .
<code>Updated: 2020-07-07</code>	

<code>\cctab_gsave_current:N</code>	<code>\cctab_gsave_current:N <category code table></code>
<code>\cctab_gsave_current:c</code>	Saves the current prevailing category codes in the <code><category code table></code> .
<code>New: 2023-05-26</code>	

33.2 Using category code tables

<code>\cctab_begin:N</code>	<code>\cctab_begin:N</code> $\langle$ category code table $\rangle$
<code>\cctab_begin:c</code>	Switches locally the category codes in force to those stored in the $\langle$ category code table $\rangle$. The prevailing codes before the function is called are added to a stack, for use with <code>\cctab_end:</code> . This function does not start a T _E X group.
Updated: 2020-07-02	
<code>\cctab_end:</code>	<code>\cctab_end:</code>
Updated: 2020-07-02	Ends the scope of a $\langle$ category code table $\rangle$ started using <code>\cctab_begin:N</code> , returning the codes to those in force before the matching <code>\cctab_begin:N</code> was used. This must be used within the same T _E X group and at the same T _E X group level as the matching <code>\cctab_begin:N</code> .
<code>\cctab_select:N</code>	<code>\cctab_select:N</code> $\langle$ category code table $\rangle$
<code>\cctab_select:c</code>	Selects the $\langle$ category code table $\rangle$ for the scope of the current group. This is in particular useful in the $\langle$ setup $\rangle$ arguments of <code>\tl_set_rescan:Nnn</code> , <code>\tl_rescan:nn</code> , <code>\cctab_const:Nn</code> , and <code>\cctab_gset:Nn</code> .
New: 2020-05-19	
Updated: 2020-07-02	
<code>\cctab_item:Nn</code> $\star$	<code>\cctab_item:Nn</code> $\langle$ category code table $\rangle$ $\{\langle$ int expr $\rangle\}$
<code>\cctab_item:cn</code> $\star$	Determines the $\langle$ character $\rangle$ with character code given by the $\langle$ int expr $\rangle$ and expands to its category code specified by the $\langle$ category code table $\rangle$.
New: 2021-05-10	

33.3 Category code table conditionals

<code>\cctab_if_exist_p:N</code> $\star$	<code>\cctab_if_exist_p:N</code> $\langle$ category code table $\rangle$
<code>\cctab_if_exist_p:c</code> $\star$	<code>\cctab_if_exist:NTF</code> $\langle$ category code table $\rangle$ $\{\langle$ true code $\rangle\}$ $\{\langle$ false code $\rangle\}$
<code>\cctab_if_exist:NTF</code> $\star$	Tests whether the $\langle$ category code table $\rangle$ is currently defined. This does not check that the $\langle$ category code table $\rangle$ really is a category code table.
<code>\cctab_if_exist:cTF</code> $\star$	

33.4 Constant and scratch category code tables

<code>\c_code_cctab</code>	Category code table for the <code>expl3</code> code environment; this does <i>not</i> include <code>@</code> , which is retained as an “other” character. Sets the <code>\endlinechar</code> value to 32 (a space).
Updated: 2020-07-10	
<code>\c_document_cctab</code>	Category code table for a standard L ^A T _E X document, as set by the L ^A T _E X kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdfT _E X <i>only</i> . No <code>babel</code> shorthands will be activated. Sets the <code>\endlinechar</code> value to 13 (normal line ending).
Updated: 2020-07-08	

\c_initex_cctab Category code table as set up by `iniTEX`.

Updated: 2020-07-02

\c_other_cctab Category code table where all characters have category code 12 (other). Sets the `\endlinechar` value to `-1`.

Updated: 2020-07-02

\c_str_cctab Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the `\endlinechar` value to `-1`.

Updated: 2020-07-02

\g_tmpa_cctab Scratch category code tables.
\g_tmpb_cctab

New: 2023-05-26

Part V

Text manipulation

Chapter 34

The `l3unicode` module

Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode *codepoints* and are designed to give usable results with both Unicode-aware and 8-bit engines.

`\codepoint_generate:nn` ★ `\codepoint_generate:nn {⟨codepoint⟩} {⟨catcode⟩}`

New: 2022-10-09
Updated: 2022-11-09

Generates one or more character tokens representing the `⟨codepoint⟩`. With Unicode engines, exactly one character token will be generated, and this will have the `⟨catcode⟩` specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the `⟨codepoint⟩`. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given `⟨catcode⟩` will be used. To allow the result of this function to be used inside an expansion context, the result is protected by `\exp_not:n`.

TeXhackers note: Users of (u)pTeX note that these engines are treated as 8-bit in this context. In particular, for upTeX, irrespective of the `\kcatcode` of the `⟨codepoint⟩`, any value outside the ASCII range will result in a series of active bytes being generated.

`\codepoint_str_generate:n` ★ `\codepoint_str_generate:n {⟨codepoint⟩}`

New: 2022-10-09

Generates one or more character tokens representing the `⟨codepoint⟩`. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the `⟨codepoint⟩`. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.

<code>\codepoint_to_category:n</code> ★	<code>\codepoint_to_category:n {⟨codepoint⟩}</code>
New: 2023-06-19	Expands to the Unicode general category identifier of the <code>⟨codepoint⟩</code>. The general category identifier is a string made up of two letter characters, the first uppercase and the second lowercase. The uppercase letters divide codepoints into broader groups, which are then refined by the lowercase letter. For example, codepoints representing letters all have identifiers starting L, for example Lu (uppercase letter), Lt (titlecase letter), etc. Full details are available in the documentation provided by the Unicode Consortium: see https://www.unicode.org/reports/tr44/#General_Category_Values
<code>\codepoint_to_nfd:n</code> ★	<code>\codepoint_to_nfd:n {⟨codepoint⟩}</code>
New: 2022-10-09	Converts the <code>⟨codepoint⟩</code> to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.

Chapter 35

The l3text module

Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the $\langle text \rangle$ are normalized and become $\{$ and $\}$, respectively.

35.1 Expanding text

<code>\text_expand:n</code>	★	<code>\text_expand:n</code>	$\{\langle text \rangle\}$

Updated: 2023-06-09

Takes user input $\langle text \rangle$ and expands the content. Protected commands (typically formatting) are left in place, and no processing of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) takes place. Commands which are neither engine- nor L^AT_EX-protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl` are excluded from expansion, as are those in `\l_text_case_exclude_arg_tl` and `\l_text_math_arg_tl`.

<code>\text_declare_expand_equivalent:Nn</code>	<code>\text_declare_expand_equivalent:Nn</code>	$\langle cmd \rangle$	$\{\langle replacement \rangle\}$
<code>\text_declare_expand_equivalent:cn</code>			

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable. A token can be “replaced” by itself if the defined replacement wraps it in `\exp_not:n`, for example

`\text_declare_expand_equivalent:Nn \' { \exp_not:n { \' } }`

35.2 Case changing

<code>\text_lowercase:n</code>	* <code>\text_uppercase:n</code> $\{\langle tokens \rangle\}$
<code>\text_uppercase:n</code>	* <code>\text_uppercase:nn</code> $\{\langle BCP-47 \rangle\}$ $\{\langle tokens \rangle\}$
<code>\text_titlecase_all:n</code>	* Takes user input $\langle text \rangle$ first applies <code>\text_expand:n</code> , then transforms the case of character tokens as specified by the function name. The category code of letters are not
<code>\text_titlecase_first:n</code>	* changed by this process when Unicode engines are used; in 8-bit engines, case changed
<code>\text_lowercase:nn</code>	* characters in the ASCII range will have the current prevailing category code, while those
<code>\text_lowercase:Vn</code>	* outside of it will be represented by active characters.
<code>\text_uppercase:nn</code>	*
<code>\text_uppercase:Vn</code>	*
<code>\text_titlecase_all:nn</code>	*
<code>\text_titlecase_all:Vn</code>	*
<code>\text_titlecase_first:nn</code>	*
<code>\text_titlecase_first:Vn</code>	*

Updated: 2023-07-08

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first *non-space* character of the $\langle tokens \rangle$ to uppercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example *ij* in Dutch which becomes *IJ*. There are two functions available for titlecasing: one which applies the change to each “word” and a second which only applies at the start of the input. (Here, “word” boundaries are spaces: at present, full Unicode word breaking is not attempted.)

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `l3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_casefold:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The first mandatory argument of commands listed in `\l_text_case_exclude_arg_tl` is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. For $\text{\pTeX}$, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the $\langle BCP-47 \rangle$ argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- **Armenian** (`hy` and `hy-x-yiwn`) The setting `hy` maps the codepoint U+0587, the ligature of letters *ech* and *yiwn*, to the codepoints for capital *ech* and *vew* when uppercasing: this follows the spelling reform which is used in Armenia. The alternative `hy-x-yiwn` maps U+0587 to capital *ech* and *yiwn* on uppercasing (also the output if Armenian is not selected at all).
- **Azeri and Turkish** (`az` and `tr`). The case pairs *I/i*-dotless and *I-dot/i* are activated for these languages. The combining dot mark is removed when lowercasing *I-dot* and introduced when upper casing *i-dotless*.

- German (**de-x-eszett**). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*.
- Greek (**el**). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant **el-x-iota** is available which converts the *ypoge-grammeni* (subscript muted iota) to capital iota when uppercasing: the standard version retains the subscript versions.
- Lithuanian (**lt**). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behavior of other accents are not modified.
- Medieval Latin (**la-x-medieval**). The characters u and v are interchanged on case changing.
- Dutch (**nl**). Capitalization of ij at the beginning of titlecased input produces IJ rather than Ij.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When `\l_text_titlecase_check_letter_bool` is **true**, codepoints which are not letters (Unicode general category L) are not changed, and only the first *letter* is uppercased. When `\l_text_titlecase_check_letter_bool` is **false**, the first codepoint is uppercased, irrespective of the general code of the character.

```
\text_declare_case_equivalent:Nn \text_declare_case_equivalent:Nn <cmd> {\replacement}
```

New: 2022-07-04

Declares that the `<replacement>` tokens should be used whenever the `<cmd>` (a single token) is encountered during case changing.

```
\text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn {\codepoint} {\replacement}
\text_declare_lowercase_mapping:nnn \text_declare_lowercase_mapping:nnn {\BCP-47} {\codepoint}
\text_declare_titlecase_mapping:nn {\replacement}
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nn
\text_declare_uppercase_mapping:nnn
```

New: 2023-04-11

Updated: 2023-04-20

Declares that the `<replacement>` tokens should be used when case mapping the `<codepoint>`, rather than the standard mapping given in the Unicode data files. The **nnn** version takes a BCP-47 tag, which can be used to specify that the customization only applies to that locale.

```
\text_declare_lowercase_exclusion:n \text_declare_lowercase_exclusion:n {\word}
\text_declare_titlecase_exclusion:n
\text_declare_uppercase_exclusion:n
```

New: 2025-06-24

Declares that the `<word>` is excluded from case changing for the appropriate operation.

<code>\text_case_switch:nnnn</code> $\star$	<code>\text_case_switch:nnnn</code> $\{\langle normal \rangle\}$ $\{\langle upper \rangle\}$ $\{\langle lower \rangle\}$ $\{\langle title \rangle\}$
New: 2022-07-04	Context-sensitive function which will expand to one of the $\langle normal \rangle$, $\langle upper \rangle$, $\langle lower \rangle$ or $\langle title \rangle$ tokens depending on the current case changing operation. Outside of case changing, the $\langle normal \rangle$ tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

35.3 Removing formatting from text

<code>\text_purify:n</code> $\star$	<code>\text_purify:n</code> $\{\langle text \rangle\}$
<code>\text_purify:(V v)</code> $\star$	Takes user input $\langle text \rangle$ and expands as described for <code>\text_expand:n</code> , then removes all functions from the resulting text. Math mode material (as delimited by pairs given in <code>\l_text_math_delims_tl</code> or as the argument to commands listed in <code>\l_text_math_arg_tl</code>) is left contained in a pair of $\$$ delimiters. Non-expandable functions present in the $\langle text \rangle$ must either have a defined equivalent (see <code>\text_declare_purify_equivalent:Nn</code>) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.
New: 2020-03-05 Updated: 2020-05-14	

<code>\text_declare_purify_equivalent:Nn</code>	<code>\text_declare_purify_equivalent:Nn</code> $\langle cmd \rangle$ $\{\langle replacement \rangle\}$
<code>\text_declare_purify_equivalent:Ne</code>	
New: 2020-03-05	

Declares that the $\langle replacement \rangle$ tokens should be used whenever the $\langle cmd \rangle$ (a single token) is encountered. The $\langle replacement \rangle$ tokens should be expandable.

35.4 Control variables

<code>\l_text_math_arg_tl</code>	Lists commands present in the $\langle text \rangle$ where the argument of the command should be treated as math mode material. The treatment here is similar to <code>\l_text_math_delims_tl</code> but for a command rather than paired delimiters.

<code>\l_text_math_delims_tl</code>	Lists pairs of tokens which delimit (in-line) math mode content; such content <i>may</i> be excluded from processing.

<code>\l_text_case_exclude_arg_tl</code>	Lists commands where the first mandatory argument is excluded from case changing.

<code>\l_text_expand_exclude_tl</code>	Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

<code>\l_text_titlecase_check_letter_bool</code>	Controls how the start of titlecasing is handled: when <code>true</code> , the first <i>letter</i> in text is considered. The standard setting is <code>true</code> .

35.5 Mapping to text

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Leading line feeds or carriage returns will be dropped due to standard T_EX processing. At present extended pictograms are not supported: these may be added in a future release. Some aspects of Indic grapheme breaking, introduced in Unicode 15, are also currently absent. In these functions, math mode is treated as a single indivisible unit, i.e. one “word” or grapheme.

`\text_map_function:nN` ☆ `\text_map_function:nN {<text>} <function>`

New: 2022-08-04
Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the `<function>`. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The `<function>` should accept one argument as *balanced text*: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_tokens:nn` ☆ `\text_map_tokens:nn {<text>} {<code>}`

New: 2025-06-22
Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the `<code>` as a trailing brace group. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The `<code>` should accept one argument as *balanced text*: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_inline:nn` `\text_map_inline:nn {<text>} {<inline function>}`

New: 2022-08-04
Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the `<inline function>`. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The `<inline function>` should consist of code which receives the grapheme as *balanced text*: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_function:nN`.

Word breaking is implemented using the standard algorithm described in Unicode Standard Annex #29. Leading line feeds or carriage returns will be dropped due to standard T_EX processing. Spaces are always considered a break even if immediately followed by an extending character. At present extended pictograms are not supported: these may be added in a future release. Language-based tailoring may be added in a future release: at present the algorithm is exactly that described in the annex.

`\text_words_map_function:nN` ☆ `\text_words_map_function:nN {<text>} {<function>}`

New: 2025-02-12

Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *words* within the result, passing each word to the `<function>`. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The `<function>` should accept one argument as `<balanced text>`: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_words_map_inline:nn`.

`\text_words_map_tokens:nn` ☆ `\text_words_map_tokens:nn {<text>} {<code>}`

New: 2025-06-22

Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *words* within the result, passing each word to the `<code>` as a trailing brace group. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The `<code>` should accept one argument as `<balanced text>`: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_words_map_inline:nn`.

`\text_words_map_inline:nn` ☆ `\text_words_map_inline:nn {<text>} {<inline function>}`

New: 2025-02-12

Updated: 2025-07-11

This takes the user input in `<text>` and expands it as with `\text_expand:n`; it then maps over the *words* within the result, passing each word to the `<function>`. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The `<inline function>` should consist of code that will accept one argument as `<balanced text>`: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_words_map_function:nN`.

`\text_map_break:` ☆ `\text_map_break:`
`\text_map_break:n` ☆ `\text_map_break:n {<code>}`

New: 2022-08-04

Used to terminate a `\text_map...` or `\text_words_map...` function before all entries in the `<text>` have been processed. This normally takes place within a conditional statement.

35.5.1 Support utilities

Implementing text functions requires some support interfaces which do not directly handle text input but are best provided in this module.

`\text_bcp_parse:n` ☆ `\text_bcp_parse:n {\langle BCP 47 \rangle}`

New: 2026-02-06

Used to parse a BCP 47 string to allow branching in text processing: see https://en.wikipedia.org/wiki/IETF_language_tag for an introduction to BCP 47 tags. The `\langle BCP 47 \rangle` input is converted to a string and parsed such that full expansion results in seven brace groups for any valid input. Other than the first brace group, these may be empty. Where multiple subtags are allowed for one type (for example variant), nested brace groups are created. The full list is

1. The language subtag, for example `en` (English) or `gsw` (Swiss German/Schweizerdeutsch); this is the only required subtag and so will always have content if the input was valid.
2. Extended language subtags: may contain up to three brace groups for these subtags, for example `en-419` will product `{419}` as the content of this subtag.
3. The script subtag, for example `Latn`
4. The region subtag, for example `GB`
5. Variant subtags: contains an open-ended list of braced groups, each containing one variant.
6. Extension subtags. The result here will comprise of the extension identifier (a single letter) followed by a brace group containing nested brace groups of each subtag. For example, the full input `gsw-u-sd-chzh` (Swiss German as used in the Canton of Zurich) would result in the extension subtag group containing `u{sd}{chzh}`.
7. Private-use subtags: contains an open-ended list of braced groups, each containing one private subtag.

Whilst the result of expansion is a token list, the parsed material is detokenized, i.e. all category code 12 (spaces are not possible). The output of the function is case folded.

Part VI

Typesetting

Chapter 36

The **l3box** module

Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a $\TeX$ group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from **l3box** are compatible with those of $\text{\LaTeX}2_{\varepsilon}$ and plain $\TeX$ and can be used interchangeably. The **l3box** commands to construct boxes, such as `\hbox:n` or `\hbox_set:Nn`, are “color-safe”, meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

36.1 Creating and initializing boxes

```
\box_new:N \box_new:N <box>
```

`\box_new:c` Creates a new `<box>` or raises an error if the name is already taken. The declaration is global. The `<box>` is initially void.

```
\box_clear:N \box_clear:N <box>
```

`\box_clear:c` Clears the content of the `<box>` by setting the box equal to `\c_empty_box`.

```
\box_gclear:N
\box_gclear:c
```

<code>\box_clear_new:N</code>	<code>\box_clear_new:N</code> $\langle box \rangle$
<code>\box_clear_new:c</code>	
<code>\box_gclear_new:N</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies
<code>\box_gclear_new:c</code>	<code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq:(cN Nc cc)</code>	
<code>\box_gset_eq:NN</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:(cN Nc cc)</code>	
<code>\box_if_exist_p:N</code>	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code>	<code>\box_if_exist:NTF</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\box_if_exist:NTF</code>	<code>\box_if_exist:NTF</code> $\star$
<code>\box_if_exist:cTF</code>	<code>\box_if_exist:cTF</code> $\star$
	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.

36.2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use:c</code>	
	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

T_EXhackers note: This is the T_EX primitive `\copy`.

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{\langle dim\ expr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_left:nn</code>	
	This function operates in vertical mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced horizontally by the given $\langle dim\ expr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .
<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{\langle dim\ expr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_down:nn</code>	
	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertically by the given $\langle dim\ expr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box\ function \rangle$ should be a box operation such as <code>\box_use:N \<box></code> or a “raw” box specification such as <code>\vbox:n { xyz }</code> .

36.3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	
	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dim\ expr \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N</code>	<code>\box</code>
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the <code>\box</code> in a form suitable for use in a <code>\dim expr</code> .	

TeXhackers note: This is the TeX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N</code>	<code>\box</code>
<code>\box_wd:c</code>	Calculates the width of the <code>\box</code> in a form suitable for use in a <code>\dim expr</code> .	

TeXhackers note: This is the TeX primitive `\wd`.

<code>\box_ht_plus_dp:N</code>	<code>\box_ht_plus_dp:N</code>	<code>\box</code>
<code>\box_ht_plus_dp:c</code>	Calculates the total vertical size (height plus depth) of the <code>\box</code> in a form suitable for use in a <code>\dim expr</code> .	

New: 2021-05-05

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn</code>	<code>\box</code>	<code>{\dim expr}</code>
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the <code>\box</code> to the value of the <code>{\dim expr}</code> .		
<code>\box_gset_dp:Nn</code>			
<code>\box_gset_dp:cn</code>			

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn</code>	<code>\box</code>	<code>{\dim expr}</code>
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the <code>\box</code> to the value of the <code>{\dim expr}</code> .		
<code>\box_gset_ht:Nn</code>			
<code>\box_gset_ht:cn</code>			

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn</code>	<code>\box</code>	<code>{\dim expr}</code>
<code>\box_set_wd:cn</code>	Set the width of the <code>\box</code> to the value of the <code>{\dim expr}</code> .		
<code>\box_gset_wd:Nn</code>			
<code>\box_gset_wd:cn</code>			

36.4 Box conditionals

<code>\box_if_empty_p:N</code>	<code>\box_if_empty_p:N</code>	<code>\box</code>
<code>\box_if_empty_p:c</code>	<code>\box_if_empty:NTF</code>	<code>\box</code> <code>{\true code}</code> <code>{\false code}</code>
<code>\box_if_empty:NTF</code>	Tests if <code>\box</code> is a empty (equal to <code>\c_empty_box</code>).	
<code>\box_if_empty:cTF</code>		

<code>\box_if_horizontal_p:N</code>	<code>\box_if_horizontal_p:N</code>	<code>\box</code>
<code>\box_if_horizontal_p:c</code>	<code>\box_if_horizontal:NTF</code>	<code>\box</code> <code>{\true code}</code> <code>{\false code}</code>
<code>\box_if_horizontal:NTF</code>	Tests if <code>\box</code> is a horizontal box.	
<code>\box_if_horizontal:cTF</code>		

<code>\box_if_vertical_p:N</code>	<code>\box_if_vertical_p:N</code>	<code>\box</code>
<code>\box_if_vertical_p:c</code>	<code>\box_if_vertical:NTF</code>	<code>\box</code> <code>{\true code}</code> <code>{\false code}</code>
<code>\box_if_vertical:NTF</code>	Tests if <code>\box</code> is a vertical box.	
<code>\box_if_vertical:cTF</code>		

36.5 The last box inserted

<code>\box_set_to_last:N</code>	<code>\box_set_to_last:N <box></code>
<code>\box_set_to_last:c</code>	
<code>\box_gset_to_last:N</code>	Sets the <code><box></code> equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the <code><box></code> is
<code>\box_gset_to_last:c</code>	always void as it is not possible to recover the last added item.

36.6 Constant boxes

<code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.

T_EXhackers note: At the T_EX level this is a void box.

36.7 Scratch boxes

<code>\l_tmpa_box</code>	Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_box</code>	

<code>\g_tmpa_box</code>	Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_box</code>	

36.8 Viewing box contents

<code>\box_show:N</code>	<code>\box_show:N <box></code>
<code>\box_show:c</code>	
	Shows full details of the content of the <code><box></code> in the terminal.
<code>\box_show:Nnn</code>	<code>\box_show:Nnn <box> {(int expr₁)} {(int expr₂)}</code>
<code>\box_show:cnn</code>	
	Display the contents of <code><box></code> in the terminal, showing the first <code><int expr₁></code> items of the box, and descending into <code><int expr₂></code> group levels.
<code>\box_log:N</code>	<code>\box_log:N <box></code>
<code>\box_log:c</code>	
	Writes full details of the content of the <code><box></code> to the log.
<code>\box_log:Nnn</code>	<code>\box_log:Nnn <box> {(int expr₁)} {(int expr₂)}</code>
<code>\box_log:cnn</code>	
	Writes the contents of <code><box></code> to the log, showing the first <code><int expr₁></code> items of the box, and descending into <code><int expr₂></code> group levels.

36.9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

36.10 Horizontal mode boxes

	<code>\hbox:n</code>	<code>\hbox:n {<contents>}</code>	
			Typesets the <code><contents></code> into a horizontal box of natural width and then includes this box in the current list for typesetting.
	<code>\hbox_to_wd:nn</code>	<code>\hbox_to_wd:nn {<dim expr>}{<contents>}</code>	
			Typesets the <code><contents></code> into a horizontal box of width <code><dim expr></code> and then includes this box in the current list for typesetting.
	<code>\hbox_to_zero:n</code>	<code>\hbox_to_zero:n {<contents>}</code>	
			Typesets the <code><contents></code> into a horizontal box of zero width and then includes this box in the current list for typesetting.
	<code>\hbox_set:Nn</code> <code>\hbox_set:cn</code> <code>\hbox_gset:Nn</code> <code>\hbox_gset:cn</code>	<code>\hbox_set:Nn <box> {<contents>}</code>	Typesets the <code><contents></code> at natural width and then stores the result inside the <code><box></code> .
	<code>\hbox_set_to_wd:Nnn</code> <code>\hbox_set_to_wd:cnn</code> <code>\hbox_gset_to_wd:Nnn</code> <code>\hbox_gset_to_wd:cnn</code>	<code>\hbox_set_to_wd:Nnn <box> {<dim expr>}{<contents>}</code>	Typesets the <code><contents></code> to the width given by the <code><dim expr></code> and then stores the result inside the <code><box></code> .
	<code>\hbox_overlap_center:n</code>	<code>\hbox_overlap_center:n {<contents>}</code>	
	New: 2020-08-25		Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.
	<code>\hbox_overlap_right:n</code>	<code>\hbox_overlap_right:n {<contents>}</code>	
			Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes to the right of the insertion point.
	<code>\hbox_overlap_left:n</code>	<code>\hbox_overlap_left:n {<contents>}</code>	
			Typesets the <code><contents></code> into a horizontal box of zero width such that material protrudes to the left of the insertion point.
	<code>\hbox_set:Nw</code> <code>\hbox_set:cw</code> <code>\hbox_set_end:</code> <code>\hbox_gset:Nw</code> <code>\hbox_gset:cw</code> <code>\hbox_gset_end:</code>	<code>\hbox_set:Nw <box> <contents> \hbox_set_end:</code>	Typesets the <code><contents></code> at natural width and then stores the result inside the <code><box></code> . In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument.

<code>\hbox_set_to_wd:Nnw</code>	<code>\hbox_set_to_wd:Nnw <box> {<dim expr>} <contents> \hbox_set_end:</code>
<code>\hbox_set_to_wd:cnw</code>	
<code>\hbox_gset_to_wd:Nnw</code>	Typesets the <code><contents></code> to the width given by the <code><dim expr></code> and then stores the result inside the <code><box></code> . In contrast to <code>\hbox_set_to_wd:Nnn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument.
<code>\hbox_gset_to_wd:cnw</code>	

<code>\hbox_unpack:N</code>	<code>\hbox_unpack:N <box></code>
<code>\hbox_unpack:c</code>	Unpacks the content of the horizontal <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

TeXhackers note: This is the TeX primitive `\unhcopy`.

36.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

<code>\vbox:n</code>	<code>\vbox:n {<contents>}</code>
	Typesets the <code><contents></code> into a vertical box of natural height and includes this box in the current list for typesetting.

<code>\vbox_top:n</code>	<code>\vbox_top:n {<contents>}</code>
	Typesets the <code><contents></code> into a vertical box of natural height and includes this box in the current list for typesetting. The vertical position of reference point of the box is equal to the baseline of the <i>first</i> item added to the box.

<code>\vbox_center:n</code>	<code>\vbox_center:n {<contents>}</code>
New: 2026-02-17	Typesets the <code><contents></code> into a vertical box of natural height and includes this box in the current list for typesetting. The vertical position of reference point of the box is at the vertical midpoint of content of the box.

<code>\vbox:w</code>	<code>\vbox:w</code>
<code>\vbox_top:w</code>	<code><contents></code>
<code>\vbox_center:w</code>	<code>\vbox_end:</code>
<code>\vbox_end:</code>	Typesets the <code><contents></code> into a vertical box of natural height and includes this box in the current list for typesetting. The reference of the box is determined as for the matching <code>:n</code> function.
New: 2026-02-25	

<code>\vbox:nn</code>	<code>\vbox_center:nn {<offset>} {<contents>}</code>

<code>\vbox_top:nn</code>	

<code>\vbox_center:nn</code>	

New: 2026-02-17	

Updated: 2026-02-24	

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting. The reference of the box is determined as for the matching `:n` function. The box is then displaced vertically by the `<offset>` (a `<dim expr>`) from existing baseline.

TeXhackers note: If the `<offset>` makes use of math font dimensions, these should be correctly initialized. In L^AT_EX, this would be achieved by applying `\check@mathfonts` immediately before using the functions. If the `<offset>` is the math axis (`\fontdimen22\textfont2`), the dimensions of the resulting box will be identical to those from `\vcenter`.

<code>\vbox:nw</code>	<code>\vbox:nw {<offset>}</code>

<code>\vbox_top:nw</code>	<code><contents></code>

<code>\vbox_center:nw</code>	<code>\vbox_end:</code>

New: 2026-02-25	

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting. The reference of the box is determined as for the matching `:n` function. The box is then displaced vertically by the `<offset>` (a `<dim expr>`) from existing baseline.

<code>\vbox_to_ht:nn</code>	<code>\vbox_to_ht:nn {<dim expr>} {<contents>}</code>

Typesets the `<contents>` into a vertical box of height `<dim expr>` and then includes this box in the current list for typesetting.

<code>\vbox_top_to_ht:nn</code>	<code>\vbox_top_to_ht:nn {<dim expr>} {<contents>}</code>

New: 2026-02-01	

Typesets the `<contents>` into a vertical box of height `<dim expr>` and then includes this box in the current list for typesetting. The vertical position of reference point of the box is equal to the baseline of the *first* item added to the box.

<code>\vbox_center_to_ht:nn</code>	<code>\vbox_center_to_ht:nn {<dim expr>} {<contents>}</code>

New: 2026-02-17	

Typesets the `<contents>` into a vertical box of height `<dim expr>` and then includes this box in the current list for typesetting. The vertical position of reference point of the box is at the vertical midpoint of content of the box.

<code>\vbox_to_ht:nw</code>	<code>\vbox_to_ht:nw {<dim expr>}</code>

<code>\vbox_top_to_ht:nw</code>	<code><contents></code>

<code>\vbox_center_to_ht:nw</code>	<code>\vbox_end:</code>

New: 2026-02-25	

Typesets the `<contents>` into a vertical box of height `<dim expr>` and includes this box in the current list for typesetting. The reference of the box is determined as for the matching `:nn` function.

<code>\vbox_to_zero:n</code>	<code>\vbox_to_zero:n {<contents>}</code>

Typesets the `<contents>` into a vertical box of zero height and then includes this box in the current list for typesetting.

<code>\vbox_set:Nn</code>	<code>\vbox_set:Nn <box> {<contents>}</code>

<code>\vbox_set:cn</code>	

<code>\vbox_gset:Nn</code>	Typesets the <code><contents></code> at natural height and then stores the result inside the <code><box></code> .

<code>\vbox_gset:cn</code>	

<code>\vbox_set_top:Nn</code>	<code>\vbox_set_top:Nn <box> {<contents>}</code>
<code>\vbox_set_top:cn</code>	Typesets the <code><contents></code> at natural height and then stores the result inside the <code><box></code> .
<code>\vbox_gset_top:Nn</code>	The baseline of the box is equal to that of the <i>first</i> item added to the box.
<code>\vbox_gset_top:cn</code>	

<code>\vbox_set_to_ht:Nnn</code>	<code>\vbox_set_to_ht:Nnn <box> {<dim expr>} {<contents>}</code>
<code>\vbox_set_to_ht:cnn</code>	Typesets the <code><contents></code> to the height given by the <code><dim expr></code> and then stores the result inside the <code><box></code> .
<code>\vbox_gset_to_ht:Nnn</code>	
<code>\vbox_gset_to_ht:cnn</code>	

<code>\vbox_set:Nw</code>	<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code>
<code>\vbox_set:cw</code>	Typesets the <code><contents></code> at natural height and then stores the result inside the <code><box></code> .
<code>\vbox_set_end:</code>	In contrast to <code>\vbox_set:Nn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument.
<code>\vbox_gset:Nw</code>	
<code>\vbox_gset:cw</code>	
<code>\vbox_gset_end:</code>	

<code>\vbox_set_to_ht:Nnw</code>	<code>\vbox_set_to_ht:Nnw <box> {<dim expr>} <contents> \vbox_set_end:</code>
<code>\vbox_set_to_ht:cnw</code>	Typesets the <code><contents></code> to the height given by the <code><dim expr></code> and then stores the result inside the <code><box></code> . In contrast to <code>\vbox_set_to_ht:Nnn</code> this function does not absorb the argument when finding the <code><content></code> , and so can be used in circumstances where the <code><content></code> may not be a simple argument.
<code>\vbox_gset_to_ht:Nnw</code>	
<code>\vbox_gset_to_ht:cnw</code>	

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn <box₁₂</code>
<code>\vbox_set_split_to_ht:(cNn Ncn ccn)</code>	
<code>\vbox_gset_split_to_ht:NNn</code>	
<code>\vbox_gset_split_to_ht:(cNn Ncn ccn)</code>	

Sets `<box1 to contain material to the height given by the <dim expr> by removing content from the top of <box2 (which must be a vertical box).`

<code>\vbox_unpack:N</code>	<code>\vbox_unpack:N <box></code>
<code>\vbox_unpack:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

TeXhackers note: This is the TeX primitive `\unvcopy`.

36.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
```

```

\group_begin:
\box_use_drop:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box

```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behavior of `drop` functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

<code>\box_use_drop:N</code>	<code>\box_use_drop:N <box></code>
<code>\box_use_drop:c</code>	

Inserts the current content of the `<box>` onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

T_EXhackers note: This is the T_EX primitive `\box`.

<code>\box_set_eq_drop:NN</code>	<code>\box_set_eq_drop:NN <box₁> <box₂></code>
<code>\box_set_eq_drop:(cN Nc cc)</code>	

Sets the content of `<box1>` equal to that of `<box2>`, then drops `<box2>`.

<code>\box_gset_eq_drop:NN</code>	<code>\box_gset_eq_drop:NN <box₁> <box₂></code>
<code>\box_gset_eq_drop:(cN Nc cc)</code>	

Sets the content of `<box1>` globally equal to that of `<box2>`, then drops `<box2>`.

<code>\hbox_unpack_drop:N</code>	<code>\hbox_unpack_drop:N <box></code>
<code>\hbox_unpack_drop:c</code>	

Unpacks the content of the horizontal `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

T_EXhackers note: This is the T_EX primitive `\unhbox`.

<code>\vbox_unpack_drop:N</code>	<code>\vbox_unpack_drop:N <box></code>
<code>\vbox_unpack_drop:c</code>	

Unpacks the content of the vertical `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

T_EXhackers note: This is the T_EX primitive `\unvbox`.

36.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing

of boxed material can best be handled by modifying boxes. These transformations are described here.

```
\box_autosize_to_wd_and_ht:Nnn \box_autosize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}
\box_autosize_to_wd_and_ht:cnn
\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
```

Resizes the $\langle\text{box}\rangle$ to fit within the given $\langle x\text{-size}\rangle$ (horizontally) and $\langle y\text{-size}\rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size}\rangle$ is the height only: it does not include any depth. The updated $\langle\text{box}\rangle$ is an hbox , irrespective of the nature of the $\langle\text{box}\rangle$ before the resizing is applied. The final size of the $\langle\text{box}\rangle$ is the smaller of $\{\langle x\text{-size}\rangle\}$ and $\{\langle y\text{-size}\rangle\}$, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the $\langle\text{box}\rangle$ to be reversed in direction, but the reference point of the $\langle\text{box}\rangle$ is unchanged. Thus a negative $\langle y\text{-size}\rangle$ results in the $\langle\text{box}\rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_autosize_to_wd_and_ht_plus_dp:Nnn \box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}
\box_autosize_to_wd_and_ht_plus_dp:cnn
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn
```

Resizes the $\langle\text{box}\rangle$ to fit within the given $\langle x\text{-size}\rangle$ (horizontally) and $\langle y\text{-size}\rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y\text{-size}\rangle$ is the total vertical size (height plus depth). The updated $\langle\text{box}\rangle$ is an hbox , irrespective of the nature of the $\langle\text{box}\rangle$ before the resizing is applied. The final size of the $\langle\text{box}\rangle$ is the smaller of $\{\langle x\text{-size}\rangle\}$ and $\{\langle y\text{-size}\rangle\}$, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the $\langle\text{box}\rangle$ to be reversed in direction, but the reference point of the $\langle\text{box}\rangle$ is unchanged. Thus a negative $\langle y\text{-size}\rangle$ results in the $\langle\text{box}\rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht:Nn \box_resize_to_ht:Nn <box> {<y-size>}
\box_resize_to_ht:cn
\box_gresize_to_ht:Nn
\box_gresize_to_ht:cn
```

Resizes the $\langle\text{box}\rangle$ to $\langle y\text{-size}\rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size}\rangle$ is a dimension expression. The $\langle y\text{-size}\rangle$ is the height only: it does not include any depth. The updated $\langle\text{box}\rangle$ is an hbox , irrespective of the nature of the $\langle\text{box}\rangle$ before the resizing is applied. A negative $\langle y\text{-size}\rangle$ causes the material in the $\langle\text{box}\rangle$ to be reversed in direction, but the reference point of the $\langle\text{box}\rangle$ is unchanged. Thus a negative $\langle y\text{-size}\rangle$ results in the $\langle\text{box}\rangle$ having a depth dependent on the height of the original and *vice versa*.

```
\box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:Nn <box> {<y-size>}
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn
```

Resizes the $\langle\text{box}\rangle$ to $\langle y\text{-size}\rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y\text{-size}\rangle$ is a dimension expression. The $\langle y\text{-size}\rangle$ is the total vertical size (height plus depth). The updated $\langle\text{box}\rangle$ is an hbox , irrespective of the nature of the $\langle\text{box}\rangle$ before the resizing is applied. A negative $\langle y\text{-size}\rangle$ causes the material in the $\langle\text{box}\rangle$ to be reversed in direction, but the reference point of the $\langle\text{box}\rangle$ is unchanged. Thus a negative $\langle y\text{-size}\rangle$ results in the $\langle\text{box}\rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>

<code>\box_resize_to_wd:cn</code>	

<code>\box_gresize_to_wd:Nn</code>	

<code>\box_gresize_to_wd:cn</code>	

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x-size \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht:Nnn</code>	<code>\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>

<code>\box_resize_to_wd_and_ht:cnn</code>	

<code>\box_gresize_to_wd_and_ht:Nnn</code>	

<code>\box_gresize_to_wd_and_ht:cnn</code>	

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_resize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}</code>

<code>\box_resize_to_wd_and_ht_plus_dp:cnn</code>	

<code>\box_gresize_to_wd_and_ht_plus_dp:Nnn</code>	

<code>\box_gresize_to_wd_and_ht_plus_dp:cnn</code>	

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>

<code>\box_rotate:cn</code>	

<code>\box_grotate:Nn</code>	

<code>\box_grotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (a $\langle fp\ expr \rangle$ in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>

<code>\box_scale:cnn</code>	

<code>\box_gscale:Nnn</code>	

<code>\box_gscale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are $\langle fp\ expr \rangle$). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-scale \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*.

36.14 Framing boxes

<code>\box_frame:Nnn</code>	<code>\box_frame:Nnn <box> {<separation>} {<thickness>}</code>
<code>\box_frame:cnn</code>	
<code>\box_gframe:Nnn</code>	Adds a frame of <code><thickness></code> (a <code><dimexpr></code>) to the <code><box></code> . The <code><separation></code> (a <code><dimexpr></code>) is left between the existing content and the frame. The updated <code><box></code> is an <code>hbox</code> , irrespective of the nature of the <code><box></code> before the frame is applied. The frame will overprint any part of the content which lies outside of the bounding box at natural size (i.e., the frame is printed “after” the content). The apparent size of the inserted box includes the frame and the border, i.e., the width of the inserted material is $\text{width}(\langle\text{content}\rangle) + 2(\langle\text{thickness}\rangle + \langle\text{separation}\rangle)$. The vertical position of the reference point of the box is unchanged by this function; the horizontal reference point is at the left of the box, i.e., the original content is displaced rightward by the <code>{<separation>}</code> and the <code>{<thickness>}</code> .
<code>\box_gframe:cnn</code>	

New: 2026-03-16

<code>\box_underline:Nnn</code>	<code>\box_underline:Nnn <box> {<separation>} {<thickness>}</code>
<code>\box_underline:cnn</code>	
<code>\box_gunderline:Nnn</code>	Adds an underline of <code><thickness></code> (a <code><dimexpr></code>) to the <code><box></code> . The <code><separation></code> (a <code><dimexpr></code>) is left between the existing content and the line. The updated <code><box></code> is an <code>hbox</code> , irrespective of the nature of the <code><box></code> before the line is applied. The underline will overprint any part of the content which lies outside of the bounding box at natural size (i.e., the line is printed “after” the content). The apparent size of the inserted box includes the underline and the border, this affects only the depth. The vertical and horizontal positions of the reference point of the box are unchanged by this function.
<code>\box_gunderline:cnn</code>	

New: 2026-03-16

36.15 Viewing part of a box

<code>\box_set_clipped:N</code>	<code>\box_set_clipped:N <box></code>
<code>\box_set_clipped:c</code>	
<code>\box_gset_clipped:N</code>	Clips the <code><box></code> in the output so that only material inside the bounding box is displayed in the output. The updated <code><box></code> is an <code>hbox</code> , irrespective of the nature of the <code><box></code> before the clipping is applied. Additional box levels are also generated by this operation.
<code>\box_gset_clipped:c</code>	

Updated: 2023-04-14

TeXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_set_trim:Nnnnn</code>	<code>\box_set_trim:Nnnnn <box> {<left>} {<bottom>} {<right>} {<top>}</code>
<code>\box_set_trim:cnnnn</code>	
<code>\box_gset_trim:Nnnnn</code>	Adjusts the bounding box of the <code><box></code> : <code><left></code> is removed from the left-hand edge of the bounding box, <code><right></code> from the right-hand edge, and so forth. All adjustments are <code><dim exprs></code> . Material outside of the bounding box is still displayed in the output unless <code>\box_set_clipped:N</code> is subsequently applied. The updated <code><box></code> is an <code>hbox</code> , irrespective of the nature of the <code><box></code> before the trim operation is applied. Additional box levels are also generated by this operation. The behavior of the operation where the trims requested is greater than the size of the box is undefined.
<code>\box_gset_trim:cnnnn</code>	

<code>\box_set_viewport:Nnnnn</code>	<code>\box_set_viewport:Nnnnn <box> {<llx>} {<lly>} {<urx>} {<ury>}</code>
<code>\box_set_viewport:cnnnn</code>	
<code>\box_gset_viewport:Nnnnn</code>	
<code>\box_gset_viewport:cnnnn</code>	Adjusts the bounding box of the <code><box></code> such that it has lower-left coordinates (<code><llx></code> , <code><lly></code>) and upper-right coordinates (<code><urx></code> , <code><ury></code>). All four coordinate positions are <code><dim exprs></code> . Material outside of the bounding box is still displayed in the output unless <code>\box_set_clipped:N</code> is subsequently applied. The updated <code><box></code> is an hbox, irrespective of the nature of the <code><box></code> before the viewport operation is applied. Additional box levels are also generated by this operation.

36.16 Primitive box conditionals

<code>\if_hbox:N</code>	<code>\if_hbox:N <box></code>
	<code><true code></code>
	<code>\else:</code>
	<code><false code></code>
	<code>\fi:</code>

Tests if `<box>` is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

<code>\if_vbox:N</code>	<code>\if_vbox:N <box></code>
	<code><true code></code>
	<code>\else:</code>
	<code><false code></code>
	<code>\fi:</code>

Tests if `<box>` is a vertical box.

T_EXhackers note: This is the T_EX primitive `\ifvbox`.

<code>\if_box_empty:N</code>	<code>\if_box_empty:N <box></code>
	<code><true code></code>
	<code>\else:</code>
	<code><false code></code>
	<code>\fi:</code>

Tests if `<box>` is an empty (void) box.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Chapter 37

The `l3coffins` module

Coffin code layer

In `expl3` terminology, a “coffin” is a box containing typeset material. Along with the box itself, the coffin structure includes information on the size and shape of the box, which makes it possible to align two or more coffins easily. This is achieved by providing a series of “poles” for each coffin. These are horizontal and vertical lines through the coffin at defined positions, for example the top or horizontal center. The points where these poles intersect are called “handles”. Two coffins can then be aligned by describing the relationship between a handle on one coffin with a handle on the second. In words, an example might then read

Align the top-left handle of coffin A with the bottom-right handle of coffin B.

The locations of coffin handles are much easier to understand visually. Figure 1 shows the standard handle positions for a coffin typeset in horizontal mode (left) and in vertical mode (right). Notice that the later case results in a greater number of handles being available. As illustrated, each handle results from the intersection of two poles. For example, the center of the coffin is marked `(hc,vc)`, i.e., it is the point of intersection of the horizontal center pole with the vertical center pole. New handles are generated automatically when poles are added to a coffin: handles are “dynamic” entities.

37.1 Controlling coffin poles

A number of standard poles are automatically generated when the coffin is set or an alignment takes place. The standard poles for all coffins are:

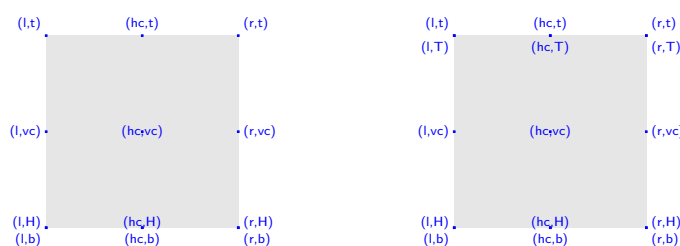


Figure 1: Standard coffin handles: left, horizontal coffin; right, vertical coffin

- `l` a pole running along the left-hand edge of the bounding box of the coffin;
- `hc` a pole running vertically through the center of the coffin half-way between the left- and right-hand edges of the bounding box (i.e., the “horizontal center”);
- `r` a pole running along the right-hand edge of the bounding box of the coffin;
- `b` a pole running along the bottom edge of the bounding box of the coffin;
- `vc` a pole running horizontally through the center of the coffin half-way between the bottom and top edges of the bounding box (i.e., the “vertical center”);
- `t` a pole running along the top edge of the bounding box of the coffin;
- `H` a pole running along the baseline of the typeset material contained in the coffin.

In addition, coffins containing vertical-mode material also feature poles which reflect the richer nature of these systems:

- `B` a pole running along the baseline of the material at the bottom of the coffin.
- `T` a pole running along the baseline of the material at the top of the coffin.

37.2 Creating and initializing coffins

<code>\coffin_new:N</code>	<code>\coffin_new:N</code> $\langle coffin \rangle$

<code>\coffin_new:c</code>	

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ is initially empty.

<code>\coffin_clear:N</code>	<code>\coffin_clear:N</code> $\langle coffin \rangle$

<code>\coffin_clear:c</code>	

<code>\coffin_gclear:N</code>	

<code>\coffin_gclear:c</code>	

Clears the content of the $\langle coffin \rangle$.

<code>\coffin_set_eq:NN</code>	<code>\coffin_set_eq:NN</code> $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

<code>\coffin_set_eq:(Nc cN cc)</code>	

<code>\coffin_gset_eq:NN</code>	

<code>\coffin_gset_eq:(Nc cN cc)</code>	

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$.

<code>\coffin_if_exist_p:N</code> $\star$	<code>\coffin_if_exist_p:N</code> $\langle coffin \rangle$

<code>\coffin_if_exist_p:c</code> $\star$	<code>\coffin_if_exist:NTF</code> $\langle coffin \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

<code>\coffin_if_exist:NTF</code> $\star$	

<code>\coffin_if_exist:cTF</code> $\star$	

Tests whether the $\langle coffin \rangle$ is currently defined.

37.3 Setting coffin content and poles

<code>\hcoffin_set:Nn</code>	<code>\hcoffin_set:Nn <coffin> {<material>}</code>
<code>\hcoffin_set:cn</code>	Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.
<code>\hcoffin_gset:Nn</code>	
<code>\hcoffin_gset:cn</code>	

<code>\hcoffin_set:Nw</code>	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code>
<code>\hcoffin_set:cw</code>	Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<code>\hcoffin_set_end:</code>	
<code>\hcoffin_gset:Nw</code>	
<code>\hcoffin_gset:cw</code>	
<code>\hcoffin_gset_end:</code>	

<code>\vcoffin_set:Nnn</code>	<code>\vcoffin_set:Nnn <coffin> {<width>} {<material>}</code>
<code>\vcoffin_set:cnn</code>	Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.
<code>\vcoffin_gset:Nnn</code>	
<code>\vcoffin_gset:cnn</code>	

Updated: 2023-02-03

<code>\vcoffin_set:Nnw</code>	<code>\vcoffin_set:Nnw <coffin> {<width>} <material> \vcoffin_set_end:</code>
<code>\vcoffin_set:cnw</code>	Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.
<code>\vcoffin_set_end:</code>	
<code>\vcoffin_gset:Nnw</code>	
<code>\vcoffin_gset:cnw</code>	
<code>\vcoffin_gset_end:</code>	

Updated: 2023-02-03

<code>\coffin_set_horizontal_pole:Nnn</code>	<code>\coffin_set_horizontal_pole:Nnn <coffin></code>
<code>\coffin_set_horizontal_pole:cn</code>	<code>{<pole>} {<offset>}</code>
<code>\coffin_gset_horizontal_pole:Nnn</code>	
<code>\coffin_gset_horizontal_pole:cn</code>	

Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the baseline of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

<code>\coffin_set_vertical_pole:Nnn</code>	<code>\coffin_set_vertical_pole:Nnn <coffin> {<pole>} {<offset>}</code>
<code>\coffin_set_vertical_pole:cn</code>	
<code>\coffin_gset_vertical_pole:Nnn</code>	
<code>\coffin_gset_vertical_pole:cn</code>	

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

<code>\coffin_reset_poles:N</code>	<code>\coffin_reset_poles:N</code> $\langle\textit{coffin}\rangle$
<code>\coffin_greset_poles:N</code>	Resets the poles of the $\langle\textit{coffin}\rangle$ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from <code>\hcoffin_set:Nn</code> or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.
New: 2023-05-17	

<code>\coffin_pole:Nn</code> ★	<code>\coffin_pole:Nn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{pole}\rangle\}$
New: 2026-02-09	Expands to four brace groups which contain the position of the given $\langle\textit{pole}\rangle$ for the $\langle\textit{coffin}\rangle$:
	1. The x coordinate of a point on the pole. 2. The y coordinate of a point on the pole. 3. The x component of a vector denoting the direction of the pole. 4. The y component of a vector denoting the direction of the pole.

The x and y coordinates are relative to the left baseline (reference point) of the coffin; the vector components are given in points with a maximum value of 1000 pt. See also the output of `\coffin_show_structure:N`.

37.4 Coffin affine transformations

<code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{width}\rangle\}$ $\{\langle\textit{total-height}\rangle\}$
<code>\coffin_resize:cnn</code>	Resized the $\langle\textit{coffin}\rangle$ to $\langle\textit{width}\rangle$ and $\langle\textit{total-height}\rangle$, both of which should be given as dimension expressions.
<code>\coffin_gresize:Nnn</code>	
<code>\coffin_gresize:cnn</code>	
<code>\coffin_rotate:Nn</code>	<code>\coffin_rotate:Nn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{angle}\rangle\}$
<code>\coffin_rotate:cn</code>	Rotates the $\langle\textit{coffin}\rangle$ by the given $\langle\textit{angle}\rangle$ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.
<code>\coffin_grotate:Nn</code>	
<code>\coffin_grotate:cn</code>	
<code>\coffin_scale:Nnn</code>	<code>\coffin_scale:Nnn</code> $\langle\textit{coffin}\rangle$ $\{\langle\textit{x-scale}\rangle\}$ $\{\langle\textit{y-scale}\rangle\}$
<code>\coffin_scale:cnn</code>	Scales the $\langle\textit{coffin}\rangle$ by a factors $\langle\textit{x-scale}\rangle$ and $\langle\textit{y-scale}\rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.
<code>\coffin_gscale:Nnn</code>	
<code>\coffin_gscale:cnn</code>	

37.5 Joining and using coffins

<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code><coffin₁> {<coffin₁-pole₁>} {<coffin₁-pole₂>}</code>
<code>\coffin_gattach:NnnNnnnn</code>	<code><coffin₂> {<coffin₂-pole₁>} {<coffin₂-pole₂>}</code>
<code>\coffin_gattach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>{<x-offset>} {<y-offset>}</code>

This function attaches `<coffin2>` to `<coffin1>` such that the bounding box of `<coffin1>` is not altered, i.e., `<coffin2>` can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating `<handle1>`, the point of intersection of `<coffin1-pole1>` and `<coffin1-pole2>`, and `<handle2>`, the point of intersection of `<coffin2-pole1>` and `<coffin2-pole2>`. `<coffin2>` is then attached to `<coffin1>` such that the relationship between `<handle1>` and `<handle2>` is described by the `<x-offset>` and `<y-offset>`. The two offsets should be given as dimension expressions.

<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code><coffin₁> {<coffin₁-pole₁>} {<coffin₁-pole₂>}</code>
<code>\coffin_gjoin:NnnNnnnn</code>	<code><coffin₂> {<coffin₂-pole₁>} {<coffin₂-pole₂>}</code>
<code>\coffin_gjoin:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>{<x-offset>} {<y-offset>}</code>

This function joins `<coffin2>` to `<coffin1>` such that the bounding box of `<coffin1>` may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating `<handle1>`, the point of intersection of `<coffin1-pole1>` and `<coffin1-pole2>`, and `<handle2>`, the point of intersection of `<coffin2-pole1>` and `<coffin2-pole2>`. `<coffin2>` is then attached to `<coffin1>` such that the relationship between `<handle1>` and `<handle2>` is described by the `<x-offset>` and `<y-offset>`. The two offsets should be given as dimension expressions.

<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn <coffin> {<pole₁>} {<pole₂>}</code>
<code>\coffin_typeset:cnnnn</code>	<code>{<x-offset>} {<y-offset>}</code>

Typesetting is carried out by first calculating `<handle>`, the point of intersection of `<pole1>` and `<pole2>`. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the `<handle>` is described by the `<x-offset>` and `<y-offset>`. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

37.6 Measuring coffins

<code>\coffin_dp:N</code>	<code>\coffin_dp:N <coffin></code>
<code>\coffin_dp:c</code>	Calculates the depth (below the baseline) of the <code><coffin></code> in a form suitable for use in a <code><dim expr></code> .

<code>\coffin_ht:N</code>	<code>\coffin_ht:N <coffin></code>
<code>\coffin_ht:c</code>	Calculates the height (above the baseline) of the <code><coffin></code> in a form suitable for use in a <code><dim expr></code> .

<code>\coffin_ht_plus_dp:N</code>	<code>\coffin_ht_plus_dp:N <coffin></code>
<code>\coffin_ht_plus_dp:c</code>	Calculates the total vertical size (height plus depth) of the <code><coffin></code> in a form suitable for use in a <code><dim expr></code> .
	New: 2024-10-01
<code>\coffin_wd:N</code>	<code>\coffin_wd:N <coffin></code>
<code>\coffin_wd:c</code>	Calculates the width of the <code><coffin></code> in a form suitable for use in a <code><dim expr></code> .

37.7 Coffin diagnostics

<code>\coffin_display_handles:Nn</code>	<code>\coffin_display_handles:Nn <coffin> {<color>}</code>
<code>\coffin_display_handles:cn</code>	This function first calculates the intersections between all of the <code><poles></code> of the <code><coffin></code> to give a set of <code><handles></code> . It then prints the <code><coffin></code> at the current location in the source, with the position of the <code><handles></code> marked on the coffin. The <code><handles></code> are labeled as part of this process: the locations of the <code><handles></code> and the labels are both printed in the <code><color></code> specified.
<code>\coffin_mark_handle:Nnnn</code>	<code>\coffin_mark_handle:Nnnn <coffin> {<pole₁>} {<pole₂>} {<color>}</code>
<code>\coffin_mark_handle:cn</code>	This function first calculates the <code><handle></code> for the <code><coffin></code> as defined by the intersection of <code><pole₁></code> and <code><pole₂></code> . It then marks the position of the <code><handle></code> on the <code><coffin></code> . The <code><handle></code> are labeled as part of this process: the location of the <code><handle></code> and the label are both printed in the <code><color></code> specified.
<code>\coffin_show_structure:N</code>	<code>\coffin_show_structure:N <coffin></code>
<code>\coffin_show_structure:c</code>	This function shows the structural information about the <code><coffin></code> in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.
	Notice that the poles of a coffin are defined by four values: the <i>x</i> and <i>y</i> coordinates of a point that the pole passes through and the <i>x</i> - and <i>y</i> -components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.
<code>\coffin_log_structure:N</code>	<code>\coffin_log_structure:N <coffin></code>
<code>\coffin_log_structure:c</code>	This function writes the structural information about the <code><coffin></code> in the log file. See also <code>\coffin_show_structure:N</code> which displays the result in the terminal.
<code>\coffin_show:N</code>	<code>\coffin_show:N <coffin></code>
<code>\coffin_show:c</code>	<code>\coffin_log:N <coffin></code>
<code>\coffin_log:N</code>	Shows full details of poles and contents of the <code><coffin></code> in the terminal or log file. See
<code>\coffin_log:c</code>	<code>\coffin_show_structure:N</code> and <code>\box_show:N</code> to show separately the pole structure and the contents.
	New: 2021-05-11

<code>\coffin_show:Nnn</code>	<code>\coffin_show:Nnn</code> $\langle coffin \rangle$ $\{ \langle int\ expr_1 \rangle \}$ $\{ \langle int\ expr_2 \rangle \}$
<code>\coffin_show:cnn</code>	<code>\coffin_log:Nnn</code> $\langle coffin \rangle$ $\{ \langle int\ expr_1 \rangle \}$ $\{ \langle int\ expr_2 \rangle \}$
<code>\coffin_log:Nnn</code>	Shows poles and contents of the $\langle coffin \rangle$ in the terminal or log file, showing the first $\langle int\ expr_1 \rangle$ items in the coffin, and descending into $\langle int\ expr_2 \rangle$ group levels. See <code>\coffin_show_structure:N</code> and <code>\box_show:Nnn</code> to show separately the pole structure and the contents.
<code>\coffin_log:cnn</code>	

New: 2021-05-11

37.8 Constants and variables

`\c_empty_coffin` A permanently empty coffin.

`\l_tmpa_coffin` Scratch coffins for local assignment. These are never used by the kernel code, and so
`\l_tmpb_coffin` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_coffin` Scratch coffins for global assignment. These are never used by the kernel code, and so
`\g_tmpb_coffin` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by
other non-kernel code and so should only be used for short-term storage.

Chapter 38

The l3color module

Color support

38.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

<code>\color_group_begin:</code>	<code>\color_group_begin:</code>
<code>\color_group_end:</code>	<code>...</code>
	<code>\color_group_end:</code>

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

<code>\color_ensure_current:</code>	<code>\color_ensure_current:</code>

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

38.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are “native” to l3color. Core models use real numbers in the range $[0, 1]$ to represent values. The core models supported here are

- **gray** Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- **rgb** Red-green-blue color, with three axes, one for each of the components

- **cmyk** Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- **hsb** Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness
- **Hsb** Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness
- **HSB** Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness
- **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number
- **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- **oklch** Lightness-chromacity-hue color in the Oklab color space (<https://bottosson.github.io/posts/oklab>), which models human perception, with three axes, real values in the range $[0, 1]$ for lightness, real values in the range $[0, 0.4]$ for chromacity and real values in the range $[0, 360]$ for hue⁸
- **oklab** Oklab color, with three axes, real values in the range $[0, 1]$ for lightness and real values in the range $[-0.4, 0.4]$ for the position on the green/red- and yellow/blue-axes⁸
- **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

Finally, there are a small number of models which are parsed to allow data transfer from **xcolor** but which should not be used by end-users. These are

- **cmy** Cyan-magenta-yellow color with three axes, one for each of the components; converted to **cmyk**
- **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness; converted to **rgb** using the standard tuning map defined by **xcolor**
- **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

⁸Be aware, that for now, this is an input model only. Color blending will not benefit from Oklab color space. It is advised to only input colors inside the sRGB gamut, i.e., mapping to $[0, 1]^3$ in the core **rgb** model. Other colors will be crudely clipped to fit inside. Thus, for most hues and lightnesses, the chroma should actually remain below 0.2.

To allow parsing of data from `xcolor`, any leading model up the first `:` will be discarded; the approach of selecting an internal form for data is *not* used in `l3color`.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 38.9 for more detail of color support for additional models.

When color is selected by model, the $\langle \text{value}(s) \rangle$ given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the $\text{\LaTeX}\ 2_{\epsilon}$ `xcolor` package and in the *PostScript Language Reference Manual*, published by Addison–Wesley.

38.3 Color expressions

In addition to allowing specification of color by model and values, `l3color` also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a value in the range 0–100. The value should be given between `!` symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50 % red and 50 % green. A trailing value is interpreted as implicitly followed by `!white`, and so

```
red!25
```

specifies 25 % red mixed with 75 % white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50 % red and 50 % of cyan *expressed in `rgb`*. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is “swapped” internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, `.` may be used to represent the color active for typesetting (the current color). This allows for example

```
.!50
```

to mean a mixture of 50 % of current color with white.

(Color expressions supported here are a subset of those provided by the $\text{\LaTeX 2}_{\epsilon}$ `xcolor` package. At present, only such features as are clearly useful have been added here.)

38.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta` and `yellow` have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

```
\color_set:nn \color_set:nn {<name>} {<color expression>}
```

Evaluates the `<color expression>` and stores the resulting color specification as the `<name>`.

```
\color_set:nnn \color_set:nnn {<name>} {<model(s)>} {<value(s)>}
```

Updated: 2024-12-24 Stores the color specification equivalent to the `<model(s)>` and `<value(s)>` as the `<name>`. The `<value(s)>` are expanded before parsing.

```
\color_set_eq:nn \color_set_eq:nn {<name1>} {<name2>}
```

Copies the color specification in `<name2>` to `<name1>`. The special name `.` may be used to represent the current color, allowing it to be saved to a name.

```
\color_if_exist_p:n * \color_if_exist_p:n {<name>}
```

```
\color_if_exist:nTF * \color_if_exist:nTF {<name>} {<true code>} {<false code>}
```

New: 2022-08-12 Tests whether `<name>` is currently defined to provide a color specification.

```
\color_show:n \color_show:n {<name>}
```

```
\color_log:n \color_log:n {<name>}
```

New: 2021-05-11 Displays the color specification stored in the `<name>` on the terminal or log file.

38.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialized functions such as fill and stroke selectors do *not* adjust this value.

<code>\color_select:n</code>	<code>\color_select:n {<color expression>}</code>
<code>\color_select:V</code>	Parses the <code><color expression></code> and then activates the resulting color specification for typeset material.
Updated: 2024-12-24	

<code>\color_select:nn</code>	<code>\color_select:nn {<model(s)>} {<value(s)>}</code>
<code>\color_select:(nV Vn VV)</code>	Activates the color specification equivalent to the <code><model(s)></code> and <code><value(s)></code> for typeset material. The <code><value(s)></code> are fully expanded before parsing.

<code>\l_color_fixed_model_tl</code>	When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

38.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, `dvips`/`dvisvgm` do not support this, and so color will need to be contained within a scope, such as `\draw_begin:/\draw_end:.`

<code>\color_fill:n</code>	<code>\color_fill:n {<color expression>}</code>
<code>\color_stroke:n</code>	Parses the <code><color expression></code> and then activates the resulting color specification for filling or stroking.

<code>\color_fill:nn</code>	<code>\color_fill:nn {<model(s)>} {<value(s)>}</code>
<code>\color_stroke:nn</code>	Activates the color specification equivalent to the <code><model(s)></code> and <code><value(s)></code> for filling or stroking. The <code><value(s)></code> are fully expanded before parsing.
Updated: 2024-12-24	

<code>color.sc</code>	When using <code>dvips</code> , this PostScript variable holds the stroke color.

38.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

<code>\color_math:nn</code>	<code>\color_math:nn {<color expression>} {<content>}</code>
<code>\color_math:nnn</code>	<code>\color_math:nnn {<model(s)>} {<value(s)>} {<content>}</code>
New: 2022-01-26	Works as for <code>\color_select:n(n)</code> but applies color only to the math mode <code><content></code> .
Updated: 2024-12-24	The <code><value(s)></code> are fully expanded before parsing. The function does not generate a group and the <code><content></code> therefore retains its math atom states. Sub/superscripts are also properly handled.

<code>\l_color_math_active_tl</code>	This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.
New: 2022-01-26	

38.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using `/` characters (as for the similar function in `xcolor`). For example, to save a color with explicit `cmymk` and `rgb` values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

38.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of `expl3`
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color

- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

<code>\color_export:nnN</code>	<code>\color_export:nnN {<color expression>} {<format>} <t1 var></code>

Parses the `<color expression>` as described earlier, then converts to the `<format>` specified and assigns the data to the `<t1 var>`.

<code>\color_export:nnnN</code>	<code>\color_export:nnnN {<model>} {<value(s)>} {<format>} <t1 var></code>

Updated: 2024-12-24 Expresses the combination of `<model>` and `<value(s)>` in an internal representation, then converts to the `<format>` specified and assigns the data to the `<t1 var>`. The `<value(s)>` are fully expanded before parsing.

38.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see <https://helpx.adobe.com/indesign/using/spot-process-colors.html> for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (**DeviceCMYK**, **DeviceRGB**, **DeviceGray**), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that **l3color** uses the shorter names **cmyk**, etc.)

<code>\color_model_new:nnn</code>	<code>\color_model_new:nnn {<model>} {<family>} {<params>}</code>

Creates a new `<model>` which is derived from the color model `<family>`. The latter should be one of

- **DeviceN**
- **ICCBased**
- **Separation**

(The `<family>` may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the `<family>`, one or more `<params>` are mandatory or optional.

For a **Separation** space, there are three *compulsory* keys.

- **name** The name of the Separation, for example the formal name of a spot color ink. Such a `<name>` may contain spaces, etc., which are not permitted in the `<model>`.
- **alternative-model** An alternative device colorspace, one of **cmyk**, **rgb**, **gray** or **CIELAB**. The three parameter-based models work as described above; see below for details of CIELAB colors.
- **alternative-values** A comma-separated list of values appropriate to the **alternative-model**. This information is used by the PDF application if the **Separation** is not available.

CIELAB color separations are created using the `alternative-model = CIELAB` setting. These colors must also have an `illuminant` key, one of `a`, `c`, `e`, `d50`, `d55`, `d65` or `d75`. The `alternative-values` in this case are the three parameters L^* , a^* and b^* of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the `colorspace` package.

CIELAB colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, `l3color` will fallback to using black as the colorant in any mixing.

For a `DeviceN` space, there is one *compulsory* key.

- **names** The names of the components of the `DeviceN` space. Each should be either the `<name>` of a `Separation` model, a process color name (`cyan`, etc.) or the special name `none`.

For a `ICCBased` space, there is one *compulsory* key.

- **file** The name of the file containing the profile.

38.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardize color which is otherwise device-dependent.

<code>\color_profile_apply:nn</code>	<code>\color_profile_apply:nn {<profile>} {<model>}</code>

New: 2021-02-23

This function applies a `<profile>` to one of the device `<models>`. The profile will then apply to all color of the selected `<model>`. The `<profile>` should specify an ICC profile file. The `<model>` has to be one the standard device models: `cmym`, `gray` or `rgb`.

Chapter 39

The l3graphics module

Graphics inclusion support

39.1 Graphics keys

Inclusion of graphic files requires a range of low-level data be passed to the backend. This is set up using a small number of key–value settings, which are stored in the **graphics** tree.

decodearray Array to decode color in bitmap graphic: when non-empty, this should be in the form of one, two or three pairs of real numbers in the range $[0, 1]$, separated by spaces.

draft Switch to enable draft mode: graphics are read but not included when this is true.

interpolate Switch which indicates whether interpolation should be applied to bitmap graphic files.

page The page to extract from a multi-page graphic file: used for **.pdf** files which may contain multiple pages.

pdf-attr Additional PDF-focussed attributes: available to allow control of extended **.pdf** structures beyond those needed for graphic inclusion. Due to backend restrictions, this key is only functional with direct PDF mode (pdfTeX and LuaTeX).

pagebox The nature of the page box setting used to determine the bounding box of material: used for **.pdf** files which feature multiple page box specifications. A choice from **art**, **bleed**, **crop**, **media**, **trim**. The standard setting is **crop**.

type The type of graphic file being included: if this key is not set, the *type* is determined from the file extension.

39.2 Including graphics

<code>\graphics_include:nn</code>	<code>\graphics_include:nn {<keys>} {<file>}</code>
<code>\graphics_include:nV</code>	Horizontal-mode command which includes the <code><file></code> as a graphic at the current location. The file <code><type></code> may be given as one of the <code><keys></code> , or will otherwise be determined from file extension. The <code><keys></code> is used to pass settings as detailed above.
New: 2025-03-14	

<code>\l_graphics_ext_type_prop</code>	Defines mapping between file extensions and file types; where there is no entry for an extension, the type is assumed to be the extension with the leading <code>.</code> removed. Entries should be made in lower case, and the key should be an extension including the leading <code>.</code> , for example
New: 2025-03-14	

```
\prop_put:Nnn \l_graphics_ext_type_prop { .ps } { eps }
```

<code>\l_graphics_search_ext_seq</code>	Extensions to use for graphic searching when the given <code><file></code> name is not found by <code>\graphics_get_full_name:nn</code> .
New: 2025-03-14	

<code>\l_graphics_search_path_seq</code>
New: 2025-03-14

Each entry is the path to a directory which should be searched when seeking a graphic file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

39.3 Utility functions

<code>\graphics_get_full_name:nN</code>	<code>\graphics_get_full_name:nN {<file>} <tl var></code>
<code>\graphics_get_full_name:nNTF</code>	<code>\graphics_get_full_name:nNTF {<file>} <tl var> {<true code>} {<false code>}</code>
New: 2025-03-14	

Searches for `<file>` first as given and then using the extensions listed in `\l_graphics_search_ext_seq`. The search path used will be the entries of `\l_graphics_search_path_seq`. If found, the full file name including any path and extension will be returned in the `<tl var>`. In the non-branching version, the `<tl var>` will be set to `\q_no_value` in the case that the graphics is not found.

<code>\graphics_get_pagecount:nN</code>	<code>\graphics_get_pagecount:nn {<file>} <tl var></code>
New: 2025-03-14	Reads the graphics <code><file></code> and extracts the number of pages, which are stored in the <code><tl var></code> .

39.4 Showing and logging included graphics

<code>\graphics_show_list:</code>	<code>\graphics_show_list:</code>
<code>\graphics_log_list:</code>	<code>\graphics_log_list:</code>

New: 2025-03-14

These functions list all graphic files loaded in a similar manner to `\file_show_list:` and `\file_log_list:`. While `\graphics_show_list:` displays the list in the terminal, `\graphics_log_list:` outputs it to the log file only. In both cases, only graphics loaded by `!3graphics` are listed.

Chapter 40

The l3opacity module Opacity (transparency) support

40.1 Selecting opacity

Opacity (transparency) shares many characteristics with color. However, limitations in terms of backends mean that it is not always possible to use a dedicated stack for tracking opacity. The best results when breaking pages are therefore likely to result using direct PDF output (pdfTeX, LuaTeX).

For users of PostScript-based routes, note that there are security restrictions which can prevent opacity being available in output. In particular, using Adobe Distiller, you will need to enable transparency in the (text-based) configuration: this is not selectable from the GUI.

For users of PDF-based routes, note that opacity only takes effect if `\RequirePackage{pdfmanagement}` or `\DocumentMetadata{}` is added *before* `\documentclass`, which loads and activates the PDF management. See `pdfmanagement-testphase.pdf` for more info.

<code>\opacity_select:n</code>	<code>\opacity_select:n {<expression>}</code>
<code>\opacity_fill:n</code>	Evaluates the <code><expression></code> , which should yield a value in the range $[0, 1]$. This is then activated as an opacity for all cases (select) or selectivity (fill and stroke). The scope of the opacity setting is limited to the current TeX group.
<code>\opacity_stroke:n</code>	

New: 2025-03-27

<code>\opacity_begin:n</code>	<code>\opacity_begin:n {<expression>}</code>
<code>\opacity_fill_begin:n</code>	<code><content></code>
<code>\opacity_stroke_begin:n</code>	<code>\opacity_end:</code>
<code>\opacity_end:</code>	Evaluates the <code><expression></code> and activates opacity for all cases (begin) or selectivity (fill and stroke). Each begin must have a matching end , but these may occur at <i>different</i> group levels.
<code>\opacity_fill_end:</code>	
<code>\opacity_stroke_end:</code>	

New: 2026-01-14

TeXhackers note: These functions do *not* create a TeX group, so can be split for example across `\halign` cells.

Chapter 41

The l3pdf module Core PDF support

41.1 Objects

41.1.1 Named objects

An $\langle object \rangle$ name should fully expand to tokens suitable for use in a label-like context.

<code>\pdf_object_new:n</code>	<code>\pdf_object_new:n {$\langle object \rangle$}</code>

New: 2022-08-23	Declares $\langle object \rangle$ as a PDF object. The object may be referenced from this point on, and written later using <code>\pdf_object_write:nnn</code> .

<code>\pdf_object_write:nnn</code>	<code>\pdf_object_write:nnn {$\langle object \rangle$} {$\langle type \rangle$} {$\langle content \rangle$}</code>

<code>\pdf_object_write:nne</code>	Writes the $\langle content \rangle$ as content of the $\langle object \rangle$. Depending on the $\langle type \rangle$ declared for the object, the format required for $\langle content \rangle$ will vary:

New: 2022-08-23

`array` A space-separated list of values

`dict` Key-value pairs in the form `/ $\langle key \rangle$   $\langle value \rangle$`

`fstream` Two brace groups: $\langle file name \rangle$ and $\langle file content \rangle$

`stream` Two brace groups: $\langle attributes dictionary \rangle$ and $\langle stream contents \rangle$

<code>\pdf_object_ref:n *</code>	<code>\pdf_object_ref:n {$\langle object \rangle$}</code>

New: 2021-02-10	Inserts the appropriate information to reference the $\langle object \rangle$ in for example page resource allocation. If the $\langle object \rangle$ does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

<code>\pdf_object_if_exist_p:n *</code>	<code>\pdf_object_if_exist_p:n {$\langle object \rangle$}</code>

<code>\pdf_object_if_exist:nTF *</code>	<code>\pdf_object_if_exist:nTF {$\langle object \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>

New: 2020-05-15	Tests whether an object with name $\{ \langle object \rangle \}$ has been defined.

41.1.2 Indexed objects

Objects can also be created using a pair of $\langle class \rangle$ and $index$; the $\langle class \rangle$ argument should expand to character tokens, whilst the $\langle index \rangle$ is an $\langle int\ expr \rangle$ and starts at 1. For large families of objects, this approach is more efficient than using individual names.

```
\pdf_object_new_indexed:nn \pdf_object_new_indexed:nn {\langle class \rangle} {\langle index \rangle}
```

New: 2024-04-01 Declares a PDF object of $\langle class \rangle$ and $\langle index \rangle$. The object may be referenced from this point on, and written later using $\backslash pdf\_object\_write\_indexed:nnnn$.

```
\pdf_object_write_indexed:nnnn \pdf_object_write_indexed:nnnn {\langle class \rangle} {\langle index \rangle} {\langle type \rangle} {\langle content \rangle}
\pdf_object_write_indexed:nnne
```

New: 2024-04-01

Writes the $\langle content \rangle$ as content of the object of $\langle class \rangle$ and $\langle index \rangle$. Depending on the $\langle type \rangle$ declared for the object, the format required for the $\langle content \rangle$ will vary

array A space-separated list of values

dict Key-value pairs in the form $/\langle key \rangle \langle value \rangle$

fstream Two brace groups: $\langle file\ name \rangle$ and $\langle file\ content \rangle$

stream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle stream\ contents \rangle$

```
\pdf_object_ref_indexed:nn * \pdf_object_ref_indexed:nn {\langle class \rangle} {\langle index \rangle}
```

New: 2024-04-01

Inserts the appropriate information to reference the object of $\langle class \rangle$ and $\langle index \rangle$ in for example page resource allocation. If the $\langle class \rangle/\langle index \rangle$ combination does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

41.1.3 General functions

```
\pdf_object_unnamed_write:nn \pdf_object_unnamed_write:nn {\langle type \rangle} {\langle content \rangle}
\pdf_object_unnamed_write:ne
```

New: 2021-02-10

Writes the $\langle content \rangle$ as content of an anonymous object. Depending on the $\langle type \rangle$, the format required for $\langle content \rangle$ will vary:

array A space-separated list of values

dict Key-value pairs in the form $/\langle key \rangle \langle value \rangle$

fstream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle file\ name \rangle$

stream Two brace groups: $\langle attributes\ (dictionary) \rangle$ and $\langle stream\ contents \rangle$

<code>\pdf_object_ref_last: *</code>	<code>\pdf_object_ref_last:</code>
New: 2021-02-10	Inserts the appropriate information to reference the last <code>\object</code> created. This is particularly useful for anonymous objects.
<code>\pdf_pageobject_ref:n *</code>	<code>\pdf_pageobject_ref:n {\abspage}</code>
New: 2021-02-10	Inserts the appropriate information to reference the <code>\abspage</code> ; the latter is expanded
Updated: 2024-04-22	fully before further processing.

41.2 Version

<code>\pdf_version_compare_p:Nn *</code>	<code>\pdf_version_compare_p:Nn <relation> {\version}</code>
<code>\pdf_version_compare:NnTF *</code>	<code>\pdf_version_compare:NnTF <relation> {\version} {\true code} {\false code}</code>
New: 2021-02-10	
	Compares the version of the PDF being created with the <code>\version</code> string specified, using the <code><relation></code> . Either the <code><true code></code> or <code><false code></code> will be left in the output stream.
<code>\pdf_version_gset:n</code>	<code>\pdf_version_gset:n {\version}</code>
<code>\pdf_version_min_gset:n</code>	
New: 2021-02-10	Sets the <code>\version</code> of the PDF being created. The min version will not alter the output version unless it is currently lower than the <code>\version</code> requested.
	This function may only be used up to the point where the PDF file is initialized. With dvips it sets <code>\pdf_version_major:</code> and <code>\pdf_version_minor:</code> and allows to compare the values with <code>\pdf_version_compare:Nn</code> , but the PDF version itself still has to be set with the command line option <code>-dCompatibilityLevel</code> of <code>ps2pdf</code> .
<code>\pdf_version:</code>	<code>\pdf_version:</code>
<code>\pdf_version_major:</code>	<code>*</code>
<code>\pdf_version_minor:</code>	<code>*</code>
New: 2021-02-10	Expands to the currently-active PDF version.
	With dvips, the PDF version is initialized to -1.-1. With dvipdfmx, it is initialized to 1.7 in releases since 2025 June, following the default T _E X Live 2025 setting; and 1.5 in previous releases.

41.3 Page (media) size

<code>\pdf_pagesize_gset:nn</code>	<code>\pdf_pagesize_gset:nn {\width} {\height}</code>
New: 2023-01-14	Sets the page size (mediabox) of the PDF being created to the <code>\width</code> and <code>\height</code> , both of which are <code><dimexpr></code> . The page size can only be set at the start of the output with dvips; with other backends, this can be adjusted on a per-page basis.

41.4 Compression

<code>\pdf_uncompress:</code>	<code>\pdf_uncompress:</code>
New: 2021-02-10	Disables any compression of the PDF, where possible.
	This function may only be used up to the point where the PDF file is initialized.

41.5 Destinations

Destinations are the places a link jumped to. Unlike the name may suggest, they don't describe an exact location in the PDF. Instead, a destination contains a reference to a page along with an instruction how to display this page. The normally used “XYZ *top left zoom*” for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

<code>\pdf_destination:nn</code>	<code>\pdf_destination:nn {<name>} {<type or integer>}</code>
New: 2021-01-03	This creates a destination. {<type or integer>} can be one of <code>fit</code> , <code>fith</code> , <code>fitv</code> , <code>fitb</code> , <code>fitbh</code> , <code>fitbv</code> , <code>fitr</code> , <code>xyz</code> or an integer representing a scale factor in percent. <code>fitr</code> here gives only a lightweight version of <code>/FitR</code> : The backend code defines <code>fitr</code> so that it will with pdfL ^A T _E X and LuaL ^A T _E X use the coordinates of the surrounding box, with dvips and dvipdfmx it falls back to <code>fit</code> . For full control use <code>\pdf_destination:nnnn</code> .

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
<code>fit</code>	<code>/Fit</code>	Fits the page to the window
<code>fith</code>	<code>/FitH top</code>	Fits the width of the page to the window
<code>fitv</code>	<code>/FitV left</code>	Fits the height of the page to the window
<code>fitb</code>	<code>/FitB</code>	Fits the page bounding box to the window
<code>fitbh</code>	<code>/FitBH top</code>	Fits the width of the page bounding box to the window.
<code>fitbv</code>	<code>/FitBV left</code>	Fits the height of the page bounding box to the window.
<code>fitr</code>	<code>/FitR left bottom right top</code>	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
<code>xyz</code>	<code>/XYZ left top null</code>	Sets a coordinate but doesn't change the zoom.
<code>{<integer>}</code>	<code>/XYZ left top zoom</code>	Sets a coordinate and a zoom meaning {<integer>}%.

<code>\pdf_destination:nnnn</code>	<code>\pdf_destination:nnnn {<name>} {<width>} {<height>} {<depth>}</code>
New: 2021-01-17	This creates a destination with <code>/FitR</code> type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

Part VII
Utilities

Chapter 42

The `\benchmark` module

Benchmarking

42.1 Benchmark

<code>\g_benchmark_duration_target_fp</code>
New: 2025-03-17

This variable (default value: 1) controls roughly for how long `\benchmark:n` will repeat code to more accurately benchmark it. The actual duration of one call to `\benchmark:n` typically lasts between half and twice `\g_benchmark_duration_target_fp` seconds, unless of course running the code only once already lasts longer than this.

<code>\g_benchmark_time_fp</code> <code>\g_benchmark_ops_fp</code>	These variables store the results of the most recently run benchmark. <code>\g_benchmark_time_fp</code> stores the time TeX took in seconds, and <code>\g_benchmark_ops_fp</code> stores the estimated number of elementary operations. The latter is not set by <code>\benchmark_tic:/\benchmark_toc:</code> .
New: 2025-03-17	

<code>\benchmark_once:n</code> <code>\benchmark_once_silent:n</code>	<code>\benchmark_once_silent:n {<code>}</code> <code>\benchmark_once:n {<code>}</code>
New: 2025-03-17	Determines the time <code>\g_benchmark_time_fp</code> (in seconds) taken by TeX to run the <code><code></code> , and an estimated number <code>\g_benchmark_ops_fp</code> of elementary operations. In addition, <code>\benchmark_once:n</code> prints these values to the terminal. The <code><code></code> is run only once so the time may be quite inaccurate for fast code.

<code>\benchmark:n</code> <code>\benchmark_silent:n</code>	<code>\benchmark:n {<code>}</code>
New: 2025-03-17	Determines the time <code>\g_benchmark_time_fp</code> (in seconds) taken by TeX to run the <code><code></code> , and an estimated number <code>\g_benchmark_ops_fp</code> of elementary operations. In addition, <code>\benchmark:n</code> prints these values to the terminal. The <code><code></code> may be run many times and not within a group, thus code with side-effects may cause problems.

`\benchmark_tic:` `\benchmark_tic: <slow code> \benchmark_toc:`

`\benchmark_toc:`

New: 2025-03-17

When it is not possible to run `\benchmark:n` (e.g., the code is part of the execution of a package which cannot be looped) the tic/toc commands can be used instead to time between two points in the code. When executed, `\benchmark_tic:` will print a line to the terminal, and `\benchmark_toc:` will print a matching line with a time to indicate the duration between them in seconds. These commands can be nested.

Part VIII
Implementation

Chapter 43

l3bootstrap implementation

```
1 <*code>
2 <@@=kernel>
```

43.1 The `\pdfstrcmp` primitive in $\text{\TeX}$

Only pdf $\text{\TeX}$ has a primitive called `\pdfstrcmp`. The $\text{\TeX}$ version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdf $\text{\TeX}$ name is “safe”.

```
3 \begingroup\expandafter\expandafter\expandafter\endgroup
4 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
5 \let\pdfstrcmp\strcmp
6 \fi
```

43.2 Loading support Lua code

When Lua $\text{\TeX}$ is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
7 \begingroup\expandafter\expandafter\expandafter\endgroup
8 \expandafter\ifx\csname directlua\endcsname\relax
9 \else
10 \ifnum\luatexversion<110 %
11 \else
```

For Lua $\text{\TeX}$ we make sure the basic support is loaded: this is only necessary in plain.

```
12 \begingroup\expandafter\expandafter\expandafter\endgroup
13 \expandafter\ifx\csname newcatcodetable\endcsname\relax
14 \input{ltluatex}%
15 \fi
16 \begingroup\expandafter\expandafter\expandafter\endgroup
17 \expandafter\ifx\csname newluabytecode\endcsname\relax
18 \else
19 \newluabytecode{@expl@luadata@bytecode
20 \fi
21 \directlua{require("expl3")}%
```


As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```

22 \ifnum 0%
23 \directlua{
24   if status.ini_version then
25     tex.write("1")
26   end
27 }>0 %
28 \everyjob\expandafter{%
29   \the\expandafter\everyjob
30   \csname\detokenize{lua_now:n}\endcsname{require("expl3")}}%
31 }%
32 \fi
33 \fi
34 \fi

```

43.3 Engine requirements

The code currently requires ε -TeX, the set of “pdfTeX extensions” *including* `\expanded`, and for Unicode engines the ability to generate arbitrary character tokens by expansion. That is covered by all supported engines since TeX Live 2019, which we therefore use as a baseline for engine and L^ATeX format support. For LuaTeX, we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since LuaTeX 1.10, which again is the one in TeX Live 2019. (u)pTeX only gained `\ifincsname` for TeX Live 2020, but at present that primitive is unused in expl3 so for the present it’s not tested. If and when that changes, we will need to revisit the code here.

```

35 \begingroup
36 \def\next{\endgroup}%
37 \def\ShortText{Required primitives not found}%
38 \def\LongText%
39 {%
40   The L3 programming layer requires the e-TeX primitives and the
41   \LineBreak 'pdfTeX utilities' as described in the README file.
42   \LineBreak
43   These are available in the engines\LineBreak
44   - pdfTeX v1.40.20\LineBreak
45   - XeTeX v0.999991\LineBreak
46   - LuaTeX v1.10\LineBreak
47   - e-(u)pTeX v3.8.2\LineBreak
48   - Prote (2021)\LineBreak
49   or later.\LineBreak
50   \LineBreak
51 }%
52 \ifnum0%
53 \expandafter\ifx\csname luatexversion\endcsname\relax
54 \expandafter\ifx\csname expanded\endcsname\relax\else 1\fi
55 \else
56 \ifnum\luatexversion<110 \else 1\fi
57 \fi
58 =0 %
59 \newlinechar'\^^J %

```

```

60     \def\LineBreak{\noexpand\MessageBreak}%
61     \expandafter\ifx\csname PackageError\endcsname\relax
62     \def\LineBreak{^^J}%
63     \begingroup
64     \lccode'\~=' \lccode'\}=' \ %
65     \lccode'\T=' \T\lccode'\H=' \H%
66     \catcode'\ =11 %
67 \lowercase{\endgroup\def\PackageError#1#2#3{%
68 \begingroup\errorcontextlines=1\immediate\write0{}\errhelp{#3}\def%
69 \
69     {#1 Error: #2.^^J^^J
70 Type H <return> for immediate help}\def~{\errmessage{%
71 \
71     }}~\endgroup}}%
72 \fi
73 \edef\next
74 {%
75     \noexpand\PackageError{expl3}{\ShortText}
76     {\LongText Loading of expl3 will abort!}%
77     \endgroup
78     \noexpand\endinput
79     }%
80 \fi
81 \next

```

43.4 The L^AT_EX3 code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied \ExplSyntaxOff becomes a “do nothing” command until \ExplSyntaxOn is used.

```

82 \protected\edef\ExplSyntaxOff
83 {%
84     \protected\def\noexpand\ExplSyntaxOff{}%
85     \catcode 9 = \the\catcode 9\relax
86     \catcode 32 = \the\catcode 32\relax
87     \catcode 34 = \the\catcode 34\relax
88     \catcode 58 = \the\catcode 58\relax
89     \catcode 94 = \the\catcode 94\relax
90     \catcode 95 = \the\catcode 95\relax
91     \catcode 124 = \the\catcode 124\relax
92     \catcode 126 = \the\catcode 126\relax
93     \endlinechar = \the\endlinechar\relax
94     \chardef\csname\detokenize{1__kernel_expl_bool}\endcsname = 0\relax
95     }%

```

(End of definition for \ExplSyntaxOff. This function is documented on page 10.)

The code environment is now set up.

```

96 \catcode 9 = 9\relax
97 \catcode 32 = 9\relax
98 \catcode 34 = 12\relax
99 \catcode 58 = 11\relax
100 \catcode 94 = 7\relax
101 \catcode 95 = 11\relax

```

```

102 \catcode 124 = 12\relax
103 \catcode 126 = 10\relax
104 \endlinechar = 32\relax

```

\l__kernel_expl_bool The status for code syntax: this is on at present.

```

105 \global\chardef\l__kernel_expl_bool = 1\relax

```

(End of definition for \l__kernel_expl_bool.)

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```

106 \protected \def \ExplSyntaxOn
107 {
108   \bool_if:NF \l__kernel_expl_bool
109   {
110     \cs_set_protected:Npe \ExplSyntaxOff
111     {
112       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
113       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
114       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
115       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
116       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
117       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
118       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
119       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
120       \tex_endlinechar:D =
121       \tex_the:D \tex_endlinechar:D \scan_stop:
122       \bool_set_false:N \l__kernel_expl_bool
123       \cs_set_protected:Npn \ExplSyntaxOff { }
124     }
125   }
126   \char_set_catcode_ignore:n { 9 } % tab
127   \char_set_catcode_ignore:n { 32 } % space
128   \char_set_catcode_other:n { 34 } % double quote
129   \char_set_catcode_letter:n { 58 } % colon
130   \char_set_catcode_math_superscript:n { 94 } % circumflex
131   \char_set_catcode_letter:n { 95 } % underscore
132   \char_set_catcode_other:n { 124 } % pipe
133   \char_set_catcode_space:n { 126 } % tilde
134   \tex_endlinechar:D = 32 \scan_stop:
135   \bool_set_true:N \l__kernel_expl_bool
136 }

```

(End of definition for \ExplSyntaxOn. This function is documented on page 10.)

```

137 </code>

```

Chapter 44

l3names implementation

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@`.

```
138 <@@=kernel>
```

```
139 <*code>
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
140 \let \tex_global:D \global
```

```
141 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `\__kernel_primitive:NN` trapped.

```
142 \begingroup
```

`\__kernel_primitive:NN` A temporary function to actually do the renaming.

```
143 \long \def \__kernel_primitive:NN #1#2
```

```
144 { \tex_global:D \tex_let:D #2 #1 }
```

(End of definition for __kernel_primitive:NN.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
145 </code>
```

```
146 <*names|code>
```

In the current incarnation of this module, all T_EX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
147 \__kernel_primitive:NN \
```

```
\tex_space:D
```

```
148 \__kernel_primitive:NN \
```

```
\tex_italiccorrection:D
```

```
149 \__kernel_primitive:NN \-
```

```
\tex_hyphen:D
```

Now all the other primitives.

```
150 \__kernel_primitive:NN \above
```

```
\tex_above:D
```

```
151 \__kernel_primitive:NN \abovedisplayshortskip
```

```
\tex_abovedisplayshortskip:D
```

```
152 \__kernel_primitive:NN \abovedisplayskip
```

```
\tex_abovedisplayskip:D
```

```
153 \__kernel_primitive:NN \abovewithdelims
```

```
\tex_abovewithdelims:D
```

```
154 \__kernel_primitive:NN \accent
```

```
\tex_accent:D
```

155	_kernel_primitive:NN	\adjdemerits	\tex_adjdemerits:D
156	_kernel_primitive:NN	\advance	\tex_advance:D
157	_kernel_primitive:NN	\afterassignment	\tex_afterassignment:D
158	_kernel_primitive:NN	\aftergroup	\tex_aftergroup:D
159	_kernel_primitive:NN	\atop	\tex_atop:D
160	_kernel_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
161	_kernel_primitive:NN	\badness	\tex_badness:D
162	_kernel_primitive:NN	\baselineskip	\tex_baselineskip:D
163	_kernel_primitive:NN	\batchmode	\tex_batchmode:D
164	_kernel_primitive:NN	\begingroup	\tex_begingroup:D
165	_kernel_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
166	_kernel_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
167	_kernel_primitive:NN	\binoppenalty	\tex_binoppenalty:D
168	_kernel_primitive:NN	\botmark	\tex_botmark:D
169	_kernel_primitive:NN	\box	\tex_box:D
170	_kernel_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
171	_kernel_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
172	_kernel_primitive:NN	\catcode	\tex_catcode:D
173	_kernel_primitive:NN	\char	\tex_char:D
174	_kernel_primitive:NN	\chardef	\tex_chardef:D
175	_kernel_primitive:NN	\cleaders	\tex_cleaders:D
176	_kernel_primitive:NN	\closein	\tex_closein:D
177	_kernel_primitive:NN	\closeout	\tex_closeout:D
178	_kernel_primitive:NN	\clubpenalty	\tex_clubpenalty:D
179	_kernel_primitive:NN	\copy	\tex_copy:D
180	_kernel_primitive:NN	\count	\tex_count:D
181	_kernel_primitive:NN	\countdef	\tex_countdef:D
182	_kernel_primitive:NN	\cr	\tex_cr:D
183	_kernel_primitive:NN	\crcr	\tex_crcr:D
184	_kernel_primitive:NN	\csname	\tex_csname:D
185	_kernel_primitive:NN	\day	\tex_day:D
186	_kernel_primitive:NN	\deadcycles	\tex_deadcycles:D
187	_kernel_primitive:NN	\def	\tex_def:D
188	_kernel_primitive:NN	\defaultthyphenchar	\tex_defaultthyphenchar:D
189	_kernel_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
190	_kernel_primitive:NN	\delcode	\tex_delcode:D
191	_kernel_primitive:NN	\delimiter	\tex_delimiter:D
192	_kernel_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
193	_kernel_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
194	_kernel_primitive:NN	\dimen	\tex_dimen:D
195	_kernel_primitive:NN	\dimendef	\tex_dimendef:D
196	_kernel_primitive:NN	\discretionary	\tex_discretionary:D
197	_kernel_primitive:NN	\displayindent	\tex_displayindent:D
198	_kernel_primitive:NN	\displaylimits	\tex_displaylimits:D
199	_kernel_primitive:NN	\displaystyle	\tex_displaystyle:D
200	_kernel_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
201	_kernel_primitive:NN	\displaywidth	\tex_displaywidth:D
202	_kernel_primitive:NN	\divide	\tex_divide:D
203	_kernel_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
204	_kernel_primitive:NN	\dp	\tex_dp:D
205	_kernel_primitive:NN	\dump	\tex_dump:D
206	_kernel_primitive:NN	\edef	\tex_edef:D
207	_kernel_primitive:NN	\else	\tex_else:D
208	_kernel_primitive:NN	\emergencystretch	\tex_emergencystretch:D

209	_kernel_primitive:NN	\end	\tex_end:D
210	_kernel_primitive:NN	\endcsname	\tex_endcsname:D
211	_kernel_primitive:NN	\endgroup	\tex_endgroup:D
212	_kernel_primitive:NN	\endinput	\tex_endinput:D
213	_kernel_primitive:NN	\endlinechar	\tex_endlinechar:D
214	_kernel_primitive:NN	\eqno	\tex_eqno:D
215	_kernel_primitive:NN	\errhelp	\tex_errhelp:D
216	_kernel_primitive:NN	\errmessage	\tex_errmessage:D
217	_kernel_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
218	_kernel_primitive:NN	\errorstopmode	\tex_errorstopmode:D
219	_kernel_primitive:NN	\escapechar	\tex_escapechar:D
220	_kernel_primitive:NN	\everycr	\tex_everycr:D
221	_kernel_primitive:NN	\everydisplay	\tex_everydisplay:D
222	_kernel_primitive:NN	\everyhbox	\tex_everyhbox:D
223	_kernel_primitive:NN	\everyjob	\tex_everyjob:D
224	_kernel_primitive:NN	\everymath	\tex_everymath:D
225	_kernel_primitive:NN	\everypar	\tex_everypar:D
226	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
227	_kernel_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
228	_kernel_primitive:NN	\expandafter	\tex_expandafter:D
229	_kernel_primitive:NN	\fam	\tex_fam:D
230	_kernel_primitive:NN	\fi	\tex_fi:D
231	_kernel_primitive:NN	\finalhyphendemerits	\tex_finalhyphendemerits:D
232	_kernel_primitive:NN	\firstmark	\tex_firstmark:D
233	_kernel_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
234	_kernel_primitive:NN	\font	\tex_font:D
235	_kernel_primitive:NN	\fontdimen	\tex_fontdimen:D
236	_kernel_primitive:NN	\fontname	\tex_fontname:D
237	_kernel_primitive:NN	\futurelet	\tex_futurelet:D
238	_kernel_primitive:NN	\gdef	\tex_gdef:D
239	_kernel_primitive:NN	\global	\tex_global:D
240	_kernel_primitive:NN	\globaldefs	\tex_globaldefs:D
241	_kernel_primitive:NN	\halign	\tex_halign:D
242	_kernel_primitive:NN	\hangafter	\tex_hangafter:D
243	_kernel_primitive:NN	\hangindent	\tex_hangindent:D
244	_kernel_primitive:NN	\hbadness	\tex_hbadness:D
245	_kernel_primitive:NN	\hbox	\tex_hbox:D
246	_kernel_primitive:NN	\hfil	\tex_hfil:D
247	_kernel_primitive:NN	\hfill	\tex_hfill:D
248	_kernel_primitive:NN	\hfilneg	\tex_hfilneg:D
249	_kernel_primitive:NN	\hfuzz	\tex_hfuzz:D
250	_kernel_primitive:NN	\hoffset	\tex_hoffset:D
251	_kernel_primitive:NN	\holdinginserts	\tex_holdinginserts:D
252	_kernel_primitive:NN	\hrule	\tex_hrule:D
253	_kernel_primitive:NN	\hsize	\tex_hsize:D
254	_kernel_primitive:NN	\hskip	\tex_hskip:D
255	_kernel_primitive:NN	\hss	\tex_hss:D
256	_kernel_primitive:NN	\ht	\tex_ht:D
257	_kernel_primitive:NN	\hyphenation	\tex_hyphenation:D
258	_kernel_primitive:NN	\hyphenchar	\tex_hyphenchar:D
259	_kernel_primitive:NN	\hyphenpenalty	\tex_hyphenpenalty:D
260	_kernel_primitive:NN	\if	\tex_if:D
261	_kernel_primitive:NN	\ifcase	\tex_ifcase:D
262	_kernel_primitive:NN	\ifcat	\tex_ifcat:D

263	_kernel_primitive:NN \ifdim	\tex_ifdim:D
264	_kernel_primitive:NN \ifeof	\tex_ifeof:D
265	_kernel_primitive:NN \iffalse	\tex_iffalse:D
266	_kernel_primitive:NN \ifhbox	\tex_ifhbox:D
267	_kernel_primitive:NN \ifhmode	\tex_ifhmode:D
268	_kernel_primitive:NN \ifinner	\tex_ifinner:D
269	_kernel_primitive:NN \ifmmode	\tex_ifmmode:D
270	_kernel_primitive:NN \ifnum	\tex_ifnum:D
271	_kernel_primitive:NN \ifodd	\tex_ifodd:D
272	_kernel_primitive:NN \iftrue	\tex_iftrue:D
273	_kernel_primitive:NN \ifvbox	\tex_ifvbox:D
274	_kernel_primitive:NN \ifvmode	\tex_ifvmode:D
275	_kernel_primitive:NN \ifvoid	\tex_ifvoid:D
276	_kernel_primitive:NN \ifx	\tex_ifx:D
277	_kernel_primitive:NN \ignorespaces	\tex_ignorespaces:D
278	_kernel_primitive:NN \immediate	\tex_immediate:D
279	_kernel_primitive:NN \indent	\tex_indent:D
280	_kernel_primitive:NN \input	\tex_input:D
281	_kernel_primitive:NN \inputlineno	\tex_inputlineno:D
282	_kernel_primitive:NN \insert	\tex_insert:D
283	_kernel_primitive:NN \insertpenalties	\tex_insertpenalties:D
284	_kernel_primitive:NN \interlinepenalty	\tex_interlinepenalty:D
285	_kernel_primitive:NN \jobname	\tex_jobname:D
286	_kernel_primitive:NN \kern	\tex_kern:D
287	_kernel_primitive:NN \language	\tex_language:D
288	_kernel_primitive:NN \lastbox	\tex_lastbox:D
289	_kernel_primitive:NN \lastkern	\tex_lastkern:D
290	_kernel_primitive:NN \lastpenalty	\tex_lastpenalty:D
291	_kernel_primitive:NN \lastskip	\tex_lastskip:D
292	_kernel_primitive:NN \lccode	\tex_lccode:D
293	_kernel_primitive:NN \leaders	\tex_leaders:D
294	_kernel_primitive:NN \left	\tex_left:D
295	_kernel_primitive:NN \lefthyphenmin	\tex_lefthyphenmin:D
296	_kernel_primitive:NN \leftskip	\tex_leftskip:D
297	_kernel_primitive:NN \leqno	\tex_leqno:D
298	_kernel_primitive:NN \let	\tex_let:D
299	_kernel_primitive:NN \limits	\tex_limits:D
300	_kernel_primitive:NN \linepenalty	\tex_linepenalty:D
301	_kernel_primitive:NN \lineskip	\tex_lineskip:D
302	_kernel_primitive:NN \lineskiplimit	\tex_lineskiplimit:D
303	_kernel_primitive:NN \long	\tex_long:D
304	_kernel_primitive:NN \looseness	\tex_looseness:D
305	_kernel_primitive:NN \lower	\tex_lower:D
306	_kernel_primitive:NN \lowercase	\tex_lowercase:D
307	_kernel_primitive:NN \mag	\tex_mag:D
308	_kernel_primitive:NN \mark	\tex_mark:D
309	_kernel_primitive:NN \mathaccent	\tex_mathaccent:D
310	_kernel_primitive:NN \mathbin	\tex_mathbin:D
311	_kernel_primitive:NN \mathchar	\tex_mathchar:D
312	_kernel_primitive:NN \mathchardef	\tex_mathchardef:D
313	_kernel_primitive:NN \mathchoice	\tex_mathchoice:D
314	_kernel_primitive:NN \mathclose	\tex_mathclose:D
315	_kernel_primitive:NN \mathcode	\tex_mathcode:D
316	_kernel_primitive:NN \mathinner	\tex_mathinner:D

317	_kernel_primitive:NN	\mathop	\tex_mathop:D
318	_kernel_primitive:NN	\mathopen	\tex_mathopen:D
319	_kernel_primitive:NN	\mathord	\tex_mathord:D
320	_kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
321	_kernel_primitive:NN	\mathrel	\tex_mathrel:D
322	_kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
323	_kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
324	_kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
325	_kernel_primitive:NN	\meaning	\tex_meaning:D
326	_kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
327	_kernel_primitive:NN	\message	\tex_message:D
328	_kernel_primitive:NN	\mkern	\tex_mkern:D
329	_kernel_primitive:NN	\month	\tex_month:D
330	_kernel_primitive:NN	\moveleft	\tex_moveleft:D
331	_kernel_primitive:NN	\moveright	\tex_moveright:D
332	_kernel_primitive:NN	\mskip	\tex_mskip:D
333	_kernel_primitive:NN	\multiply	\tex_multiply:D
334	_kernel_primitive:NN	\muskip	\tex_muskip:D
335	_kernel_primitive:NN	\muskipdef	\tex_muskipdef:D
336	_kernel_primitive:NN	\newlinechar	\tex_newlinechar:D
337	_kernel_primitive:NN	\noalign	\tex_noalign:D
338	_kernel_primitive:NN	\noboundary	\tex_noboundary:D
339	_kernel_primitive:NN	\noexpand	\tex_noexpand:D
340	_kernel_primitive:NN	\noindent	\tex_noindent:D
341	_kernel_primitive:NN	\nolimits	\tex_nolimits:D
342	_kernel_primitive:NN	\nonscript	\tex_nonscript:D
343	_kernel_primitive:NN	\nonstopmode	\tex_nonstopmode:D
344	_kernel_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D
345	_kernel_primitive:NN	\nullfont	\tex_nullfont:D
346	_kernel_primitive:NN	\number	\tex_number:D
347	_kernel_primitive:NN	\omit	\tex_omit:D
348	_kernel_primitive:NN	\openin	\tex_openin:D
349	_kernel_primitive:NN	\openout	\tex_openout:D
350	_kernel_primitive:NN	\or	\tex_or:D
351	_kernel_primitive:NN	\outer	\tex_outer:D
352	_kernel_primitive:NN	\output	\tex_output:D
353	_kernel_primitive:NN	\outputpenalty	\tex_outputpenalty:D
354	_kernel_primitive:NN	\over	\tex_over:D
355	_kernel_primitive:NN	\overfullrule	\tex_overfullrule:D
356	_kernel_primitive:NN	\overline	\tex_overline:D
357	_kernel_primitive:NN	\overwithdelims	\tex_overwithdelims:D
358	_kernel_primitive:NN	\pagedepth	\tex_pagedepth:D
359	_kernel_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
360	_kernel_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
361	_kernel_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
362	_kernel_primitive:NN	\pagegoal	\tex_pagegoal:D
363	_kernel_primitive:NN	\pageshrink	\tex_pageshrink:D
364	_kernel_primitive:NN	\pagestretch	\tex_pagestretch:D
365	_kernel_primitive:NN	\pagetotal	\tex_pagetotal:D
366	_kernel_primitive:NN	\par	\tex_par:D
367	_kernel_primitive:NN	\parfillskip	\tex_parfillskip:D
368	_kernel_primitive:NN	\parindent	\tex_parindent:D
369	_kernel_primitive:NN	\parshape	\tex_parshape:D
370	_kernel_primitive:NN	\parskip	\tex_parskip:D

371	_kernel_primitive:NN	\patterns	\tex_patterns:D
372	_kernel_primitive:NN	\pausing	\tex_pausing:D
373	_kernel_primitive:NN	\penalty	\tex_penalty:D
374	_kernel_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
375	_kernel_primitive:NN	\predisplaypenalty	\tex_predisplaypenalty:D
376	_kernel_primitive:NN	\predisplaysize	\tex_predisplaysize:D
377	_kernel_primitive:NN	\pretolerance	\tex_pretolerance:D
378	_kernel_primitive:NN	\prevdepth	\tex_prevdepth:D
379	_kernel_primitive:NN	\prevgraf	\tex_prevgraf:D
380	_kernel_primitive:NN	\radical	\tex_radical:D
381	_kernel_primitive:NN	\raise	\tex_raise:D
382	_kernel_primitive:NN	\read	\tex_read:D
383	_kernel_primitive:NN	\relax	\tex_relax:D
384	_kernel_primitive:NN	\relpenalty	\tex_relpenalty:D
385	_kernel_primitive:NN	\right	\tex_right:D
386	_kernel_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
387	_kernel_primitive:NN	\rightskip	\tex_rightskip:D
388	_kernel_primitive:NN	\romannumeral	\tex_romannumeral:D
389	_kernel_primitive:NN	\scriptfont	\tex_scriptfont:D
390	_kernel_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
391	_kernel_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
392	_kernel_primitive:NN	\scriptspace	\tex_scriptspace:D
393	_kernel_primitive:NN	\scriptstyle	\tex_scriptstyle:D
394	_kernel_primitive:NN	\scrollmode	\tex_scrollmode:D
395	_kernel_primitive:NN	\setbox	\tex_setbox:D
396	_kernel_primitive:NN	\setlanguage	\tex_setlanguage:D
397	_kernel_primitive:NN	\sfcode	\tex_sfcode:D
398	_kernel_primitive:NN	\shipout	\tex_shipout:D
399	_kernel_primitive:NN	\show	\tex_show:D
400	_kernel_primitive:NN	\showbox	\tex_showbox:D
401	_kernel_primitive:NN	\showboxbreadth	\tex_showboxbreadth:D
402	_kernel_primitive:NN	\showboxdepth	\tex_showboxdepth:D
403	_kernel_primitive:NN	\showlists	\tex_showlists:D
404	_kernel_primitive:NN	\showthe	\tex_showthe:D
405	_kernel_primitive:NN	\skewchar	\tex_skewchar:D
406	_kernel_primitive:NN	\skip	\tex_skip:D
407	_kernel_primitive:NN	\skipdef	\tex_skipdef:D
408	_kernel_primitive:NN	\spacefactor	\tex_spacefactor:D
409	_kernel_primitive:NN	\spaceskip	\tex_spaceskip:D
410	_kernel_primitive:NN	\span	\tex_span:D
411	_kernel_primitive:NN	\special	\tex_special:D
412	_kernel_primitive:NN	\splitbotmark	\tex_splitbotmark:D
413	_kernel_primitive:NN	\splitfirstmark	\tex_splitfirstmark:D
414	_kernel_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
415	_kernel_primitive:NN	\splittopskip	\tex_splittopskip:D
416	_kernel_primitive:NN	\string	\tex_string:D
417	_kernel_primitive:NN	\tabskip	\tex_tabskip:D
418	_kernel_primitive:NN	\textfont	\tex_textfont:D
419	_kernel_primitive:NN	\textstyle	\tex_textstyle:D
420	_kernel_primitive:NN	\the	\tex_the:D
421	_kernel_primitive:NN	\thickmuskip	\tex_thickmuskip:D
422	_kernel_primitive:NN	\thinmuskip	\tex_thinmuskip:D
423	_kernel_primitive:NN	\time	\tex_time:D
424	_kernel_primitive:NN	\toks	\tex_toks:D

425	<code>_kernel_primitive:NN \toksdef</code>	<code>\tex_toksdef:D</code>
426	<code>_kernel_primitive:NN \tolerance</code>	<code>\tex_tolerance:D</code>
427	<code>_kernel_primitive:NN \topmark</code>	<code>\tex_topmark:D</code>
428	<code>_kernel_primitive:NN \topskip</code>	<code>\tex_topskip:D</code>
429	<code>_kernel_primitive:NN \tracingcommands</code>	<code>\tex_tracingcommands:D</code>
430	<code>_kernel_primitive:NN \tracinglostchars</code>	<code>\tex_tracinglostchars:D</code>
431	<code>_kernel_primitive:NN \tracingmacros</code>	<code>\tex_tracingmacros:D</code>
432	<code>_kernel_primitive:NN \tracingonline</code>	<code>\tex_tracingonline:D</code>
433	<code>_kernel_primitive:NN \tracingoutput</code>	<code>\tex_tracingoutput:D</code>
434	<code>_kernel_primitive:NN \tracingpages</code>	<code>\tex_tracingpages:D</code>
435	<code>_kernel_primitive:NN \tracingparagraphs</code>	<code>\tex_tracingparagraphs:D</code>
436	<code>_kernel_primitive:NN \tracingrestores</code>	<code>\tex_tracingrestores:D</code>
437	<code>_kernel_primitive:NN \tracingstats</code>	<code>\tex_tracingstats:D</code>
438	<code>_kernel_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
439	<code>_kernel_primitive:NN \uchyph</code>	<code>\tex_uchyph:D</code>
440	<code>_kernel_primitive:NN \underline</code>	<code>\tex_underline:D</code>
441	<code>_kernel_primitive:NN \unhbox</code>	<code>\tex_unhbox:D</code>
442	<code>_kernel_primitive:NN \unhcopy</code>	<code>\tex_unhcopy:D</code>
443	<code>_kernel_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
444	<code>_kernel_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>
445	<code>_kernel_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
446	<code>_kernel_primitive:NN \unvbox</code>	<code>\tex_unvbox:D</code>
447	<code>_kernel_primitive:NN \unvcopy</code>	<code>\tex_unvcopy:D</code>
448	<code>_kernel_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
449	<code>_kernel_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
450	<code>_kernel_primitive:NN \valign</code>	<code>\tex_valign:D</code>
451	<code>_kernel_primitive:NN \vbadness</code>	<code>\tex_vbadness:D</code>
452	<code>_kernel_primitive:NN \vbox</code>	<code>\tex_vbox:D</code>
453	<code>_kernel_primitive:NN \vcenter</code>	<code>\tex_vcenter:D</code>
454	<code>_kernel_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
455	<code>_kernel_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
456	<code>_kernel_primitive:NN \vfilneg</code>	<code>\tex_vfilneg:D</code>
457	<code>_kernel_primitive:NN \vfuzz</code>	<code>\tex_vfuzz:D</code>
458	<code>_kernel_primitive:NN \voffset</code>	<code>\tex_voffset:D</code>
459	<code>_kernel_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
460	<code>_kernel_primitive:NN \vsize</code>	<code>\tex_vsize:D</code>
461	<code>_kernel_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
462	<code>_kernel_primitive:NN \vsplit</code>	<code>\tex_vsplit:D</code>
463	<code>_kernel_primitive:NN \vss</code>	<code>\tex_vss:D</code>
464	<code>_kernel_primitive:NN \vtop</code>	<code>\tex_vtop:D</code>
465	<code>_kernel_primitive:NN \wd</code>	<code>\tex_wd:D</code>
466	<code>_kernel_primitive:NN \widowpenalty</code>	<code>\tex_widowpenalty:D</code>
467	<code>_kernel_primitive:NN \write</code>	<code>\tex_write:D</code>
468	<code>_kernel_primitive:NN \xdef</code>	<code>\tex_xdef:D</code>
469	<code>_kernel_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
470	<code>_kernel_primitive:NN \xspaceskip</code>	<code>\tex_xspaceskip:D</code>
471	<code>_kernel_primitive:NN \year</code>	<code>\tex_year:D</code>

Primitives introduced by ϵ -TeX.

472	<code>_kernel_primitive:NN \beginL</code>	<code>\tex_beginL:D</code>
473	<code>_kernel_primitive:NN \beginR</code>	<code>\tex_beginR:D</code>
474	<code>_kernel_primitive:NN \botmarks</code>	<code>\tex_botmarks:D</code>
475	<code>_kernel_primitive:NN \clubpenalties</code>	<code>\tex_clubpenalties:D</code>
476	<code>_kernel_primitive:NN \currentgrouplevel</code>	<code>\tex_currentgrouplevel:D</code>
477	<code>_kernel_primitive:NN \currentgrouptype</code>	<code>\tex_currentgrouptype:D</code>

478	_kernel_primitive:NN	\currentifbranch	\tex_currentifbranch:D
479	_kernel_primitive:NN	\currentiflevel	\tex_currentiflevel:D
480	_kernel_primitive:NN	\currentifttype	\tex_currentifttype:D
481	_kernel_primitive:NN	\detokenize	\tex_detokenize:D
482	_kernel_primitive:NN	\dimexpr	\tex_dimexpr:D
483	_kernel_primitive:NN	\displaywidowpenalties	\tex_displaywidowpenalties:D
484	_kernel_primitive:NN	\endL	\tex_endL:D
485	_kernel_primitive:NN	\endR	\tex_endR:D
486	_kernel_primitive:NN	\eTeXrevision	\tex_eTeXrevision:D
487	_kernel_primitive:NN	\eTeXversion	\tex_eTeXversion:D
488	_kernel_primitive:NN	\everyeof	\tex_everyeof:D
489	_kernel_primitive:NN	\firstmarks	\tex_firstmarks:D
490	_kernel_primitive:NN	\fontchardp	\tex_fontchardp:D
491	_kernel_primitive:NN	\fontcharht	\tex_fontcharht:D
492	_kernel_primitive:NN	\fontcharic	\tex_fontcharic:D
493	_kernel_primitive:NN	\fontcharwd	\tex_fontcharwd:D
494	_kernel_primitive:NN	\glueexpr	\tex_glueexpr:D
495	_kernel_primitive:NN	\glueshrink	\tex_glueshrink:D
496	_kernel_primitive:NN	\glueshrinkorder	\tex_glueshrinkorder:D
497	_kernel_primitive:NN	\gluestretch	\tex_gluestretch:D
498	_kernel_primitive:NN	\gluestretchorder	\tex_gluestretchorder:D
499	_kernel_primitive:NN	\gluetomu	\tex_gluetomu:D
500	_kernel_primitive:NN	\ifcsname	\tex_ifcsname:D
501	_kernel_primitive:NN	\ifdefined	\tex_ifdefined:D
502	_kernel_primitive:NN	\iffontchar	\tex_iffontchar:D
503	_kernel_primitive:NN	\interactionmode	\tex_interactionmode:D
504	_kernel_primitive:NN	\interlinepenalties	\tex_interlinepenalties:D
505	_kernel_primitive:NN	\lastlinefit	\tex_lastlinefit:D
506	_kernel_primitive:NN	\lastnodetype	\tex_lastnodetype:D
507	_kernel_primitive:NN	\marks	\tex_marks:D
508	_kernel_primitive:NN	\middle	\tex_middle:D
509	_kernel_primitive:NN	\muexpr	\tex_muexpr:D
510	_kernel_primitive:NN	\mutoglu	\tex_mutoglu:D
511	_kernel_primitive:NN	\numexpr	\tex_numexpr:D
512	_kernel_primitive:NN	\pagediscards	\tex_pagediscards:D
513	_kernel_primitive:NN	\parshapedimen	\tex_parshapedimen:D
514	_kernel_primitive:NN	\parshapeindent	\tex_parshapeindent:D
515	_kernel_primitive:NN	\parshapelength	\tex_parshapelength:D
516	_kernel_primitive:NN	\predisplaydirection	\tex_predisplaydirection:D
517	_kernel_primitive:NN	\protected	\tex_protected:D
518	_kernel_primitive:NN	\readline	\tex_readline:D
519	_kernel_primitive:NN	\savinghyphcodes	\tex_savinghyphcodes:D
520	_kernel_primitive:NN	\savingvdiscards	\tex_savingvdiscards:D
521	_kernel_primitive:NN	\scantokens	\tex_scantokens:D
522	_kernel_primitive:NN	\showgroups	\tex_showgroups:D
523	_kernel_primitive:NN	\showifs	\tex_showifs:D
524	_kernel_primitive:NN	\showtokens	\tex_showtokens:D
525	_kernel_primitive:NN	\splitbotmarks	\tex_splitbotmarks:D
526	_kernel_primitive:NN	\splitdiscards	\tex_splitdiscards:D
527	_kernel_primitive:NN	\splitfirstmarks	\tex_splitfirstmarks:D
528	_kernel_primitive:NN	\TeXXeTstate	\tex_TeXXeTstate:D
529	_kernel_primitive:NN	\topmarks	\tex_topmarks:D
530	_kernel_primitive:NN	\tracingassigns	\tex_tracingassigns:D
531	_kernel_primitive:NN	\tracinggroups	\tex_tracinggroups:D

532	<code>_kernel_primitive:NN</code>	<code>\tracingifs</code>	<code>\tex_tracingifs:D</code>
533	<code>_kernel_primitive:NN</code>	<code>\tracingnesting</code>	<code>\tex_tracingnesting:D</code>
534	<code>_kernel_primitive:NN</code>	<code>\tracingscantokens</code>	<code>\tex_tracingscantokens:D</code>
535	<code>_kernel_primitive:NN</code>	<code>\unexpanded</code>	<code>\tex_unexpanded:D</code>
536	<code>_kernel_primitive:NN</code>	<code>\unless</code>	<code>\tex_unless:D</code>
537	<code>_kernel_primitive:NN</code>	<code>\widowpenalties</code>	<code>\tex_widowpenalties:D</code>

Post- ϵ -TeX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdfTeX which directly relate to PDF output: these are copied with the names unchanged.

538	<code>_kernel_primitive:NN</code>	<code>\pdfannot</code>	<code>\tex_pdfannot:D</code>
539	<code>_kernel_primitive:NN</code>	<code>\pdfcatalog</code>	<code>\tex_pdfcatalog:D</code>
540	<code>_kernel_primitive:NN</code>	<code>\pdfcompresslevel</code>	<code>\tex_pdfcompresslevel:D</code>
541	<code>_kernel_primitive:NN</code>	<code>\pdfcolorstack</code>	<code>\tex_pdfcolorstack:D</code>
542	<code>_kernel_primitive:NN</code>	<code>\pdfcolorstackinit</code>	<code>\tex_pdfcolorstackinit:D</code>
543	<code>_kernel_primitive:NN</code>	<code>\pdfdecimaldigits</code>	<code>\tex_pdfdecimaldigits:D</code>
544	<code>_kernel_primitive:NN</code>	<code>\pdfdest</code>	<code>\tex_pdfdest:D</code>
545	<code>_kernel_primitive:NN</code>	<code>\pdfdestmargin</code>	<code>\tex_pdfdestmargin:D</code>
546	<code>_kernel_primitive:NN</code>	<code>\pdfendlink</code>	<code>\tex_pdfendlink:D</code>
547	<code>_kernel_primitive:NN</code>	<code>\pdfendthread</code>	<code>\tex_pdfendthread:D</code>
548	<code>_kernel_primitive:NN</code>	<code>\pdffakespace</code>	<code>\tex_pdffakespace:D</code>
549	<code>_kernel_primitive:NN</code>	<code>\pdffontattr</code>	<code>\tex_pdffontattr:D</code>
550	<code>_kernel_primitive:NN</code>	<code>\pdffontname</code>	<code>\tex_pdffontname:D</code>
551	<code>_kernel_primitive:NN</code>	<code>\pdffontobjnum</code>	<code>\tex_pdffontobjnum:D</code>
552	<code>_kernel_primitive:NN</code>	<code>\pdfgamma</code>	<code>\tex_pdfgamma:D</code>
553	<code>_kernel_primitive:NN</code>	<code>\pdfgentounicode</code>	<code>\tex_pdfgentounicode:D</code>
554	<code>_kernel_primitive:NN</code>	<code>\pdfglyphtounicode</code>	<code>\tex_pdfglyphtounicode:D</code>
555	<code>_kernel_primitive:NN</code>	<code>\pdfhorigin</code>	<code>\tex_pdfhorigin:D</code>
556	<code>_kernel_primitive:NN</code>	<code>\pdfimageapplygamma</code>	<code>\tex_pdfimageapplygamma:D</code>
557	<code>_kernel_primitive:NN</code>	<code>\pdfimagegamma</code>	<code>\tex_pdfimagegamma:D</code>
558	<code>_kernel_primitive:NN</code>	<code>\pdfimagehicolor</code>	<code>\tex_pdfimagehicolor:D</code>
559	<code>_kernel_primitive:NN</code>	<code>\pdfimageresolution</code>	<code>\tex_pdfimageresolution:D</code>
560	<code>_kernel_primitive:NN</code>	<code>\pdfincludechars</code>	<code>\tex_pdfincludechars:D</code>
561	<code>_kernel_primitive:NN</code>	<code>\pdfinclusioncopyfonts</code>	<code>\tex_pdfinclusioncopyfonts:D</code>
562	<code>_kernel_primitive:NN</code>	<code>\pdfinclusionerrorlevel</code>	
563		<code>\tex_pdfinclusionerrorlevel:D</code>	
564	<code>_kernel_primitive:NN</code>	<code>\pdfinfo</code>	<code>\tex_pdfinfo:D</code>
565	<code>_kernel_primitive:NN</code>	<code>\pdfinfoomitdate</code>	<code>\tex_pdfinfoomitdate:D</code>
566	<code>_kernel_primitive:NN</code>	<code>\pdfinterwordsoff</code>	<code>\tex_pdfinterwordsoff:D</code>
567	<code>_kernel_primitive:NN</code>	<code>\pdfinterwordspaceon</code>	<code>\tex_pdfinterwordspaceon:D</code>
568	<code>_kernel_primitive:NN</code>	<code>\pdflastannot</code>	<code>\tex_pdflastannot:D</code>
569	<code>_kernel_primitive:NN</code>	<code>\pdflastlink</code>	<code>\tex_pdflastlink:D</code>
570	<code>_kernel_primitive:NN</code>	<code>\pdflastobj</code>	<code>\tex_pdflastobj:D</code>
571	<code>_kernel_primitive:NN</code>	<code>\pdflastxform</code>	<code>\tex_pdflastxform:D</code>
572	<code>_kernel_primitive:NN</code>	<code>\pdflastximage</code>	<code>\tex_pdflastximage:D</code>
573	<code>_kernel_primitive:NN</code>	<code>\pdflastximagecolordepth</code>	
574		<code>\tex_pdflastximagecolordepth:D</code>	
575	<code>_kernel_primitive:NN</code>	<code>\pdflastximagepages</code>	<code>\tex_pdflastximagepages:D</code>
576	<code>_kernel_primitive:NN</code>	<code>\pdflinkmargin</code>	<code>\tex_pdflinkmargin:D</code>
577	<code>_kernel_primitive:NN</code>	<code>\pdfliteral</code>	<code>\tex_pdfliteral:D</code>
578	<code>_kernel_primitive:NN</code>	<code>\pdfmapfile</code>	<code>\tex_pdfmapfile:D</code>
579	<code>_kernel_primitive:NN</code>	<code>\pdfmapline</code>	<code>\tex_pdfmapline:D</code>
580	<code>_kernel_primitive:NN</code>	<code>\pdfmajorversion</code>	<code>\tex_pdfmajorversion:D</code>

```

581 \__kernel_primitive:NN \pdfminorversion      \tex_pdfminorversion:D
582 \__kernel_primitive:NN \pdfnames             \tex_pdfnames:D
583 \__kernel_primitive:NN \pdfnobluiltintounicode \tex_pdfnobluiltintounicode:D
584 \__kernel_primitive:NN \pdfobj              \tex_pdfobj:D
585 \__kernel_primitive:NN \pdfobjcompresslevel  \tex_pdfobjcompresslevel:D
586 \__kernel_primitive:NN \pdfomitcharset      \tex_pdfomitcharset:D
587 \__kernel_primitive:NN \pdfoutline          \tex_pdfoutline:D
588 \__kernel_primitive:NN \pdfoutput           \tex_pdfoutput:D
589 \__kernel_primitive:NN \pdfpageattr         \tex_pdfpageattr:D
590 \__kernel_primitive:NN \pdfpagebox          \tex_pdfpagebox:D
591 \__kernel_primitive:NN \pdfpageref          \tex_pdfpageref:D
592 \__kernel_primitive:NN \pdfpageresources     \tex_pdfpageresources:D
593 \__kernel_primitive:NN \pdfpagesattr        \tex_pdfpagesattr:D
594 \__kernel_primitive:NN \pdfptexuseunderscore \tex_pdfptexuseunderscore:D
595 \__kernel_primitive:NN \pdfrefobj           \tex_pdfrefobj:D
596 \__kernel_primitive:NN \pdfrefxform         \tex_pdfrefxform:D
597 \__kernel_primitive:NN \pdfrefximage        \tex_pdfrefximage:D
598 \__kernel_primitive:NN \pdfrestore          \tex_pdfrestore:D
599 \__kernel_primitive:NN \pdfretval           \tex_pdfretval:D
600 \__kernel_primitive:NN \pdfrunninglinkoff   \tex_pdfrunninglinkoff:D
601 \__kernel_primitive:NN \pdfrunninglinkon    \tex_pdfrunninglinkon:D
602 \__kernel_primitive:NN \pdfsave             \tex_pdfsave:D
603 \__kernel_primitive:NN \pdfsetmatrix        \tex_pdfsetmatrix:D
604 \__kernel_primitive:NN \pdfstartlink        \tex_pdfstartlink:D
605 \__kernel_primitive:NN \pdfstartthread      \tex_pdfstartthread:D
606 \__kernel_primitive:NN \pdfsuppressptexinfo \tex_pdfsuppressptexinfo:D
607 \__kernel_primitive:NN \pdfsuppresswarningdupdest
608 \tex_pdfsuppresswarningdupdest:D
609 \__kernel_primitive:NN \pdfsuppresswarningdupmap
610 \tex_pdfsuppresswarningdupmap:D
611 \__kernel_primitive:NN \pdfsuppresswarningpagegroup
612 \tex_pdfsuppresswarningpagegroup:D
613 \__kernel_primitive:NN \pdfthread           \tex_pdfthread:D
614 \__kernel_primitive:NN \pdfthreadmargin     \tex_pdfthreadmargin:D
615 \__kernel_primitive:NN \pdftrailer          \tex_pdftrailer:D
616 \__kernel_primitive:NN \pdftrailerid        \tex_pdftrailerid:D
617 \__kernel_primitive:NN \pdfuniqueresname    \tex_pdfuniqueresname:D
618 \__kernel_primitive:NN \pdfvorigin          \tex_pdfvorigin:D
619 \__kernel_primitive:NN \pdfxform            \tex_pdfxform:D
620 \__kernel_primitive:NN \pdfxformname        \tex_pdfxformname:D
621 \__kernel_primitive:NN \pdfximage           \tex_pdfximage:D
622 \__kernel_primitive:NN \pdfximagebbox       \tex_pdfximagebbox:D

```

These are not related to PDF output and either already appear in other engines without the `\pdf` prefix, or might reasonably do so at some future stage. We therefore drop the leading `pdf` here.

```

623 \__kernel_primitive:NN \ifpdfabsdim         \tex_ifabsdim:D
624 \__kernel_primitive:NN \ifpdfabsnum         \tex_ifabsnum:D
625 \__kernel_primitive:NN \ifpdfprimitive       \tex_ifprimitive:D
626 \__kernel_primitive:NN \pdfadjustinterwordglue
627 \tex_adjustinterwordglue:D
628 \__kernel_primitive:NN \pdfadjustspacing     \tex_adjustspacing:D
629 \__kernel_primitive:NN \pdfappendkern       \tex_appendkern:D
630 \__kernel_primitive:NN \pdfcopyfont         \tex_copyfont:D

```

631	<code>__kernel_primitive:NN</code>	<code>\pdfcreationdate</code>	<code>\tex_creationdate:D</code>
632	<code>__kernel_primitive:NN</code>	<code>\pdfdraftmode</code>	<code>\tex_draftmode:D</code>
633	<code>__kernel_primitive:NN</code>	<code>\pdfeachlinedepth</code>	<code>\tex_eachlinedepth:D</code>
634	<code>__kernel_primitive:NN</code>	<code>\pdfeachlineheight</code>	<code>\tex_eachlineheight:D</code>
635	<code>__kernel_primitive:NN</code>	<code>\pdfelapsedtime</code>	<code>\tex_elapsedtime:D</code>
636	<code>__kernel_primitive:NN</code>	<code>\pdfescapehex</code>	<code>\tex_escapehex:D</code>
637	<code>__kernel_primitive:NN</code>	<code>\pdfescapename</code>	<code>\tex_escapename:D</code>
638	<code>__kernel_primitive:NN</code>	<code>\pdfescapestring</code>	<code>\tex_escapestring:D</code>
639	<code>__kernel_primitive:NN</code>	<code>\pdffirstlineheight</code>	<code>\tex_firstlineheight:D</code>
640	<code>__kernel_primitive:NN</code>	<code>\pdffontexpand</code>	<code>\tex_fontexpand:D</code>
641	<code>__kernel_primitive:NN</code>	<code>\pdffontsize</code>	<code>\tex_fontsize:D</code>
642	<code>__kernel_primitive:NN</code>	<code>\pdfignoreddimen</code>	<code>\tex_ignoreddimen:D</code>
643	<code>__kernel_primitive:NN</code>	<code>\pdfinsertht</code>	<code>\tex_insertht:D</code>
644	<code>__kernel_primitive:NN</code>	<code>\pdflastlinedepth</code>	<code>\tex_lastlinedepth:D</code>
645	<code>__kernel_primitive:NN</code>	<code>\pdflastmatch</code>	<code>\tex_lastmatch:D</code>
646	<code>__kernel_primitive:NN</code>	<code>\pdflastxpos</code>	<code>\tex_lastxpos:D</code>
647	<code>__kernel_primitive:NN</code>	<code>\pdflastypos</code>	<code>\tex_lastypos:D</code>
648	<code>__kernel_primitive:NN</code>	<code>\pdfmatch</code>	<code>\tex_match:D</code>
649	<code>__kernel_primitive:NN</code>	<code>\pdfnoligatures</code>	<code>\tex_noligatures:D</code>
650	<code>__kernel_primitive:NN</code>	<code>\pdfnormaldeviate</code>	<code>\tex_normaldeviate:D</code>
651	<code>__kernel_primitive:NN</code>	<code>\pdfpageheight</code>	<code>\tex_pageheight:D</code>
652	<code>__kernel_primitive:NN</code>	<code>\pdfpagewidth</code>	<code>\tex_pagewidth:D</code>
653	<code>__kernel_primitive:NN</code>	<code>\pdfpkmode</code>	<code>\tex_pkmode:D</code>
654	<code>__kernel_primitive:NN</code>	<code>\pdfpkresolution</code>	<code>\tex_pkresolution:D</code>
655	<code>__kernel_primitive:NN</code>	<code>\pdfprimitive</code>	<code>\tex_primitive:D</code>
656	<code>__kernel_primitive:NN</code>	<code>\pdfprependkern</code>	<code>\tex_prependkern:D</code>
657	<code>__kernel_primitive:NN</code>	<code>\pdfprotrudechars</code>	<code>\tex_protrudechars:D</code>
658	<code>__kernel_primitive:NN</code>	<code>\pdfpxdimen</code>	<code>\tex_pxdimen:D</code>
659	<code>__kernel_primitive:NN</code>	<code>\pdfrandomseed</code>	<code>\tex_randomseed:D</code>
660	<code>__kernel_primitive:NN</code>	<code>\pdfresettimer</code>	<code>\tex_resettimer:D</code>
661	<code>__kernel_primitive:NN</code>	<code>\pdfsavepos</code>	<code>\tex_savepos:D</code>
662	<code>__kernel_primitive:NN</code>	<code>\pdfsetrandomseed</code>	<code>\tex_setrandomseed:D</code>
663	<code>__kernel_primitive:NN</code>	<code>\pdfshellescape</code>	<code>\tex_shellescape:D</code>
664	<code>__kernel_primitive:NN</code>	<code>\pdftracingfonts</code>	<code>\tex_tracingfonts:D</code>
665	<code>__kernel_primitive:NN</code>	<code>\pdfunescapehex</code>	<code>\tex_unescapehex:D</code>
666	<code>__kernel_primitive:NN</code>	<code>\pdfuniformdeviate</code>	<code>\tex_uniformdeviate:D</code>

The version primitives are not related to PDF mode but are pdfTeX-specific, so again are carried forward unchanged.

667	<code>__kernel_primitive:NN</code>	<code>\pdfptxbanner</code>	<code>\tex_pdfptxbanner:D</code>
668	<code>__kernel_primitive:NN</code>	<code>\pdfptxrevision</code>	<code>\tex_pdfptxrevision:D</code>
669	<code>__kernel_primitive:NN</code>	<code>\pdfptxversion</code>	<code>\tex_pdfptxversion:D</code>

These ones appear in pdfTeX but don't have pdf in the name at all: no decisions to make.

670	<code>__kernel_primitive:NN</code>	<code>\efcode</code>	<code>\tex_efcode:D</code>
671	<code>__kernel_primitive:NN</code>	<code>\ifincsname</code>	<code>\tex_ifincsname:D</code>
672	<code>__kernel_primitive:NN</code>	<code>\knaccode</code>	<code>\tex_knaccode:D</code>
673	<code>__kernel_primitive:NN</code>	<code>\knbccode</code>	<code>\tex_knbccode:D</code>
674	<code>__kernel_primitive:NN</code>	<code>\knbscode</code>	<code>\tex_knbscode:D</code>
675	<code>__kernel_primitive:NN</code>	<code>\leftmarginkern</code>	<code>\tex_leftmarginkern:D</code>
676	<code>__kernel_primitive:NN</code>	<code>\letterspacefont</code>	<code>\tex_letterspacefont:D</code>
677	<code>__kernel_primitive:NN</code>	<code>\lpcode</code>	<code>\tex_lpcode:D</code>
678	<code>__kernel_primitive:NN</code>	<code>\quitvmode</code>	<code>\tex_quitvmode:D</code>
679	<code>__kernel_primitive:NN</code>	<code>\rightmarginkern</code>	<code>\tex_rightmarginkern:D</code>

```

680 \__kernel_primitive:NN \rprcode \tex_rprcode:D
681 \__kernel_primitive:NN \shbscode \tex_shbscode:D
682 \__kernel_primitive:NN \stbscode \tex_stbscode:D
683 \__kernel_primitive:NN \synctex \tex_synctex:D
684 \__kernel_primitive:NN \tagcode \tex_tagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

685 </names | code>
686 <*code>
687 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
688 \tex_long:D \tex_def:D \use_none:n #1 { }
689 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
690 {
691   \tex_ifdefined:D #1
692   \tex_expandafter:D \use_ii:nn
693   \tex_fi:D
694   \use_none:n { \tex_global:D \tex_let:D #2 #1 }
695 }
696 </code>
697 <*names | code>

```

Some pdfTeX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

```

698 \__kernel_primitive:NN \pdfstrcmp \tex_strcmp:D
699 \__kernel_primitive:NN \pdffilesize \tex_filesize:D
700 \__kernel_primitive:NN \pdfmdfivesum \tex_mdfivesum:D
701 \__kernel_primitive:NN \pdffilemoddate \tex_filemoddate:D
702 \__kernel_primitive:NN \pdffiledump \tex_filedump:D

```

XeTeX-specific primitives. Note that XeTeX's \strcmp is handled earlier and is “rolled up” into \pdfstrcmp. A few cross-compatibility names which lack the pdf of the original are handled later.

```

703 \__kernel_primitive:NN \suppressfontnotfounderror
704 \tex_suppressfontnotfounderror:D
705 \__kernel_primitive:NN \XeTeXcharclass \tex_XeTeXcharclass:D
706 \__kernel_primitive:NN \XeTeXcharglyph \tex_XeTeXcharglyph:D
707 \__kernel_primitive:NN \XeTeXcountfeatures \tex_XeTeXcountfeatures:D
708 \__kernel_primitive:NN \XeTeXcountglyphs \tex_XeTeXcountglyphs:D
709 \__kernel_primitive:NN \XeTeXcountselectors \tex_XeTeXcountselectors:D
710 \__kernel_primitive:NN \XeTeXcountvariations \tex_XeTeXcountvariations:D
711 \__kernel_primitive:NN \XeTeXdefaultencoding \tex_XeTeXdefaultencoding:D
712 \__kernel_primitive:NN \XeTeXdashbreakstate \tex_XeTeXdashbreakstate:D
713 \__kernel_primitive:NN \XeTeXfeaturecode \tex_XeTeXfeaturecode:D
714 \__kernel_primitive:NN \XeTeXfeaturename \tex_XeTeXfeaturename:D
715 \__kernel_primitive:NN \XeTeXfindfeaturebyname
716 \tex_XeTeXfindfeaturebyname:D
717 \__kernel_primitive:NN \XeTeXfindselectorbyname
718 \tex_XeTeXfindselectorbyname:D
719 \__kernel_primitive:NN \XeTeXfindvariationbyname
720 \tex_XeTeXfindvariationbyname:D
721 \__kernel_primitive:NN \XeTeXfirstfontchar \tex_XeTeXfirstfontchar:D

```

```

722 \__kernel_primitive:NN \XeTeXfonttype \tex_XeTeXfonttype:D
723 \__kernel_primitive:NN \XeTeXgenerateactualtext
724 \tex_XeTeXgenerateactualtext:D
725 \__kernel_primitive:NN \XeTeXglyph \tex_XeTeXglyph:D
726 \__kernel_primitive:NN \XeTeXglyphbounds \tex_XeTeXglyphbounds:D
727 \__kernel_primitive:NN \XeTeXglyphindex \tex_XeTeXglyphindex:D
728 \__kernel_primitive:NN \XeTeXglyphname \tex_XeTeXglyphname:D
729 \__kernel_primitive:NN \XeTeXinputencoding \tex_XeTeXinputencoding:D
730 \__kernel_primitive:NN \XeTeXinputnormalization
731 \tex_XeTeXinputnormalization:D
732 \__kernel_primitive:NN \XeTeXinterchartokenstate
733 \tex_XeTeXinterchartokenstate:D
734 \__kernel_primitive:NN \XeTeXinterchartoks \tex_XeTeXinterchartoks:D
735 \__kernel_primitive:NN \XeTeXisdefaultselector
736 \tex_XeTeXisdefaultselector:D
737 \__kernel_primitive:NN \XeTeXisexclusivefeature
738 \tex_XeTeXisexclusivefeature:D
739 \__kernel_primitive:NN \XeTeXlastfontchar \tex_XeTeXlastfontchar:D
740 \__kernel_primitive:NN \XeTeXlinebreakskip \tex_XeTeXlinebreakskip:D
741 \__kernel_primitive:NN \XeTeXlinebreaklocale \tex_XeTeXlinebreaklocale:D
742 \__kernel_primitive:NN \XeTeXlinebreakpenalty \tex_XeTeXlinebreakpenalty:D
743 \__kernel_primitive:NN \XeTeXOTcountfeatures \tex_XeTeXOTcountfeatures:D
744 \__kernel_primitive:NN \XeTeXOTcountlanguages \tex_XeTeXOTcountlanguages:D
745 \__kernel_primitive:NN \XeTeXOTcountscripts \tex_XeTeXOTcountscripts:D
746 \__kernel_primitive:NN \XeTeXOTfeaturetag \tex_XeTeXOTfeaturetag:D
747 \__kernel_primitive:NN \XeTeXOTlanguagetag \tex_XeTeXOTlanguagetag:D
748 \__kernel_primitive:NN \XeTeXOTscripttag \tex_XeTeXOTscripttag:D
749 \__kernel_primitive:NN \XeTeXpdf file \tex_XeTeXpdf file:D
750 \__kernel_primitive:NN \XeTeXpdfpagecount \tex_XeTeXpdfpagecount:D
751 \__kernel_primitive:NN \XeTeXpicfile \tex_XeTeXpicfile:D
752 \__kernel_primitive:NN \XeTeXrevision \tex_XeTeXrevision:D
753 \__kernel_primitive:NN \XeTeXselectorname \tex_XeTeXselectorname:D
754 \__kernel_primitive:NN \XeTeXtracingfonts \tex_XeTeXtracingfonts:D
755 \__kernel_primitive:NN \XeTeXupwardsmode \tex_XeTeXupwardsmode:D
756 \__kernel_primitive:NN \XeTeXuseglyphmetrics \tex_XeTeXuseglyphmetrics:D
757 \__kernel_primitive:NN \XeTeXvariation \tex_XeTeXvariation:D
758 \__kernel_primitive:NN \XeTeXvariationdefault \tex_XeTeXvariationdefault:D
759 \__kernel_primitive:NN \XeTeXvariationmax \tex_XeTeXvariationmax:D
760 \__kernel_primitive:NN \XeTeXvariationmin \tex_XeTeXvariationmin:D
761 \__kernel_primitive:NN \XeTeXvariationname \tex_XeTeXvariationname:D
762 \__kernel_primitive:NN \XeTeXversion \tex_XeTeXversion:D
763 \__kernel_primitive:NN \XeTeXselectorcode \tex_XeTeXselectorcode:D
764 \__kernel_primitive:NN \XeTeXinterwordspaceshaping
765 \tex_XeTeXinterwordspaceshaping:D
766 \__kernel_primitive:NN \XeTeXhyphenatablelength
767 \tex_XeTeXhyphenatablelength:D

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

768 \__kernel_primitive:NN \creationdate \tex_creationdate:D
769 \__kernel_primitive:NN \elapsedtime \tex_elapsedtime:D
770 \__kernel_primitive:NN \filedump \tex_filedump:D
771 \__kernel_primitive:NN \filemoddate \tex_filemoddate:D
772 \__kernel_primitive:NN \filesize \tex_filesize:D
773 \__kernel_primitive:NN \mdfivesum \tex_mdfivesum:D
774 \__kernel_primitive:NN \ifprimitive \tex_ifprimitive:D

```



```

775 \__kernel_primitive:NN \primitive \tex_primitive:D
776 \__kernel_primitive:NN \resettimer \tex_resettimer:D
777 \__kernel_primitive:NN \shellescape \tex_shellescape:D
778 \__kernel_primitive:NN \XeTeXprotrudechars \tex_protrudechars:D

```

Primitives from LuaTeX, some of which have been ported back to X_YTeX.

```

779 \__kernel_primitive:NN \alignmark \tex_alignmark:D
780 \__kernel_primitive:NN \aligntab \tex_aligntab:D
781 \__kernel_primitive:NN \attribute \tex_attribute:D
782 \__kernel_primitive:NN \attributedef \tex_attributedef:D
783 \__kernel_primitive:NN \automaticdiscretionary
784 \tex_automaticdiscretionary:D
785 \__kernel_primitive:NN \automatichyphenmode \tex_automatichyphenmode:D
786 \__kernel_primitive:NN \automatichyphenpenalty
787 \tex_automatichyphenpenalty:D
788 \__kernel_primitive:NN \beginsname \tex_beginsname:D
789 \__kernel_primitive:NN \bodydir \tex_bodydir:D
790 \__kernel_primitive:NN \bodydirection \tex_bodydirection:D
791 \__kernel_primitive:NN \boundary \tex_boundary:D
792 \__kernel_primitive:NN \boxdir \tex_boxdir:D
793 \__kernel_primitive:NN \boxdirection \tex_boxdirection:D
794 \__kernel_primitive:NN \breakafterdirmode \tex_breakafterdirmode:D
795 \__kernel_primitive:NN \catcodetable \tex_catcodetable:D
796 \__kernel_primitive:NN \clearmarks \tex_clearmarks:D
797 % \__kernel_primitive:NN \compoundhyphenmode
798 % \tex_compoundhyphenmode:D % not documented in manual
799 \__kernel_primitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
800 \__kernel_primitive:NN \crampedscriptscriptstyle
801 \tex_crampedscriptscriptstyle:D
802 \__kernel_primitive:NN \crampedscriptstyle \tex_crampedscriptstyle:D
803 \__kernel_primitive:NN \crampedtextstyle \tex_crampedtextstyle:D
804 \__kernel_primitive:NN \csstring \tex_csstring:D
805 \__kernel_primitive:NN \deferred \tex_deferred:D
806 \__kernel_primitive:NN \discretionaryligaturemode
807 \tex_discretionaryligaturemode:D
808 \__kernel_primitive:NN \directlua \tex_directlua:D
809 \__kernel_primitive:NN \dviextension \tex_dviextension:D
810 \__kernel_primitive:NN \dvifedback \tex_dvifedback:D
811 \__kernel_primitive:NN \dvivariable \tex_dvivariable:D
812 \__kernel_primitive:NN \eTeXglueshrinkorder \tex_eTeXglueshrinkorder:D
813 \__kernel_primitive:NN \eTeXgluestretchorder \tex_eTeXgluestretchorder:D
814 \__kernel_primitive:NN \endlocalcontrol \tex_endlocalcontrol:D
815 \__kernel_primitive:NN \etoksapp \tex_etoksapp:D
816 \__kernel_primitive:NN \etokspre \tex_etokspre:D
817 \__kernel_primitive:NN \exceptionpenalty \tex_exceptionpenalty:D
818 \__kernel_primitive:NN \exhyphenchar \tex_exhyphenchar:D
819 \__kernel_primitive:NN \explicithyphenpenalty \tex_explicithyphenpenalty:D
820 \__kernel_primitive:NN \expanded \tex_expanded:D
821 \__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
822 \__kernel_primitive:NN \firstvalidlanguage \tex_firstvalidlanguage:D
823 % \__kernel_primitive:NN \fixupboxesmode
824 % \tex_fixupboxesmode:D % not documented in manual
825 \__kernel_primitive:NN \fontid \tex_fontid:D
826 \__kernel_primitive:NN \formatname \tex_formatname:D
827 \__kernel_primitive:NN \hjcode \tex_hjcode:D

```

828	_kernel_primitive:NN	\hpack	\tex_hpack:D
829	_kernel_primitive:NN	\hyphenationbounds	\tex_hyphenationbounds:D
830	_kernel_primitive:NN	\hyphenationmin	\tex_hyphenationmin:D
831	_kernel_primitive:NN	\hyphenpenaltymode	\tex_hyphenpenaltymode:D
832	_kernel_primitive:NN	\gleaders	\tex_gleaders:D
833	_kernel_primitive:NN	\glet	\tex_glet:D
834	_kernel_primitive:NN	\glyphdimensionsmode	\tex_glyphdimensionsmode:D
835	_kernel_primitive:NN	\gtoksapp	\tex_gtoksapp:D
836	_kernel_primitive:NN	\gtokspre	\tex_gtokspre:D
837	_kernel_primitive:NN	\ifcondition	\tex_ifcondition:D
838	_kernel_primitive:NN	\immediateassigned	\tex_immediateassigned:D
839	_kernel_primitive:NN	\immediateassignment	\tex_immediateassignment:D
840	_kernel_primitive:NN	\initcatcodetable	\tex_initcatcodetable:D
841	_kernel_primitive:NN	\lastnamedcs	\tex_lastnamedcs:D
842	_kernel_primitive:NN	\latelua	\tex_latelua:D
843	_kernel_primitive:NN	\lateluafunction	\tex_lateluafunction:D
844	_kernel_primitive:NN	\leftghost	\tex_leftghost:D
845	_kernel_primitive:NN	\letcharcode	\tex_letcharcode:D
846	_kernel_primitive:NN	\linedir	\tex_linedir:D
847	_kernel_primitive:NN	\linedirection	\tex_linedirection:D
848	_kernel_primitive:NN	\localbrokenpenalty	\tex_localbrokenpenalty:D
849	_kernel_primitive:NN	\localinterlinepenalty	\tex_localinterlinepenalty:D
850	_kernel_primitive:NN	\luabytecode	\tex_luabytecode:D
851	_kernel_primitive:NN	\luabytecodecall	\tex_luabytecodecall:D
852	_kernel_primitive:NN	\luacopyinputnodes	\tex_luacopyinputnodes:D
853	_kernel_primitive:NN	\luadef	\tex_luadef:D
854	_kernel_primitive:NN	\localleftbox	\tex_localleftbox:D
855	_kernel_primitive:NN	\localrightbox	\tex_localrightbox:D
856	_kernel_primitive:NN	\luaescapestring	\tex_luaescapestring:D
857	_kernel_primitive:NN	\luafunction	\tex_luafunction:D
858	_kernel_primitive:NN	\luafunctioncall	\tex_luafunctioncall:D
859	_kernel_primitive:NN	\luatexbanner	\tex_luatexbanner:D
860	_kernel_primitive:NN	\luatexrevision	\tex_luatexrevision:D
861	_kernel_primitive:NN	\luatexversion	\tex_luatexversion:D
862	_kernel_primitive:NN	\mathdefaultsmode	\tex_mathdefaultsmode:D
863	_kernel_primitive:NN	\mathdelimitersmode	\tex_mathdelimitersmode:D
864	_kernel_primitive:NN	\mathdir	\tex_mathdir:D
865	_kernel_primitive:NN	\mathdirection	\tex_mathdirection:D
866	_kernel_primitive:NN	\mathdisplayskipmode	\tex_mathdisplayskipmode:D
867	_kernel_primitive:NN	\matheqdirmode	\tex_matheqdirmode:D
868	_kernel_primitive:NN	\matheqnogapstep	\tex_matheqnogapstep:D
869	_kernel_primitive:NN	\mathemptydisplaymode	\tex_mathemptydisplaymode:D
870	_kernel_primitive:NN	\mathflattenmode	\tex_mathflattenmode:D
871	_kernel_primitive:NN	\mathitalicsmode	\tex_mathitalicsmode:D
872	_kernel_primitive:NN	\mathnolimitsmode	\tex_mathnolimitsmode:D
873	_kernel_primitive:NN	\mathoption	\tex_mathoption:D
874	_kernel_primitive:NN	\mathpenaltiesmode	\tex_mathpenaltiesmode:D
875	_kernel_primitive:NN	\mathrulesfam	\tex_mathrulesfam:D
876	% _kernel_primitive:NN	\mathrulesmode	
877	% \tex_mathrulesmode:D	% not documented in manual	
878	% _kernel_primitive:NN	\mathrulethicknessmode	
879	% \tex_mathrulethicknessmode:D	% not documented in manual	
880	_kernel_primitive:NN	\mathscriptsmode	\tex_mathscriptsmode:D
881	_kernel_primitive:NN	\mathscriptboxmode	\tex_mathscriptboxmode:D

882	<code>_kernel_primitive:NN \mathscriptcharmode</code>	<code>\tex_mathscriptcharmode:D</code>
883	<code>_kernel_primitive:NN \mathstyle</code>	<code>\tex_mathstyle:D</code>
884	<code>_kernel_primitive:NN \mathsurroundmode</code>	<code>\tex_mathsurroundmode:D</code>
885	<code>_kernel_primitive:NN \mathsurroundskip</code>	<code>\tex_mathsurroundskip:D</code>
886	<code>_kernel_primitive:NN \nohrule</code>	<code>\tex_nohrule:D</code>
887	<code>_kernel_primitive:NN \nokerns</code>	<code>\tex_nokerns:D</code>
888	<code>_kernel_primitive:NN \noligs</code>	<code>\tex_noligs:D</code>
889	<code>_kernel_primitive:NN \nospaces</code>	<code>\tex_nospaces:D</code>
890	<code>_kernel_primitive:NN \novrule</code>	<code>\tex_novrule:D</code>
891	<code>_kernel_primitive:NN \outputbox</code>	<code>\tex_outputbox:D</code>
892	<code>_kernel_primitive:NN \pagebottomoffset</code>	<code>\tex_pagebottomoffset:D</code>
893	<code>_kernel_primitive:NN \pagedir</code>	<code>\tex_pagedir:D</code>
894	<code>_kernel_primitive:NN \pagedirection</code>	<code>\tex_pagedirection:D</code>
895	<code>_kernel_primitive:NN \pageleftoffset</code>	<code>\tex_pageleftoffset:D</code>
896	<code>_kernel_primitive:NN \pagerightoffset</code>	<code>\tex_pagerightoffset:D</code>
897	<code>_kernel_primitive:NN \pagetopoffset</code>	<code>\tex_pagetopoffset:D</code>
898	<code>_kernel_primitive:NN \pardir</code>	<code>\tex_pardir:D</code>
899	<code>_kernel_primitive:NN \pardirection</code>	<code>\tex_pardirection:D</code>
900	<code>_kernel_primitive:NN \pdfextension</code>	<code>\tex_pdfextension:D</code>
901	<code>_kernel_primitive:NN \pdffeedback</code>	<code>\tex_pdffeedback:D</code>
902	<code>_kernel_primitive:NN \pdfvariable</code>	<code>\tex_pdfvariable:D</code>
903	<code>_kernel_primitive:NN \postexhyphenchar</code>	<code>\tex_postexhyphenchar:D</code>
904	<code>_kernel_primitive:NN \posthyphenchar</code>	<code>\tex_posthyphenchar:D</code>
905	<code>_kernel_primitive:NN \prebinoppenalty</code>	<code>\tex_prebinoppenalty:D</code>
906	<code>_kernel_primitive:NN \predisplaygapfactor</code>	<code>\tex_predisplaygapfactor:D</code>
907	<code>_kernel_primitive:NN \preexhyphenchar</code>	<code>\tex_preexhyphenchar:D</code>
908	<code>_kernel_primitive:NN \prehyphenchar</code>	<code>\tex_prehyphenchar:D</code>
909	<code>_kernel_primitive:NN \prerelpenalty</code>	<code>\tex_prerelpenalty:D</code>
910	<code>_kernel_primitive:NN \protrusionboundary</code>	<code>\tex_protrusionboundary:D</code>
911	<code>_kernel_primitive:NN \rightghost</code>	<code>\tex_rightghost:D</code>
912	<code>_kernel_primitive:NN \savecatcodetable</code>	<code>\tex_savecatcodetable:D</code>
913	<code>_kernel_primitive:NN \scantexttokens</code>	<code>\tex_scantexttokens:D</code>
914	<code>_kernel_primitive:NN \setfontid</code>	<code>\tex_setfontid:D</code>
915	<code>_kernel_primitive:NN \shapemode</code>	<code>\tex_shapemode:D</code>
916	<code>_kernel_primitive:NN \suppressifcsnameerror</code>	<code>\tex_suppressifcsnameerror:D</code>
917	<code>_kernel_primitive:NN \suppresslongerror</code>	<code>\tex_suppresslongerror:D</code>
918	<code>_kernel_primitive:NN \suppressmathparerror</code>	<code>\tex_suppressmathparerror:D</code>
919	<code>_kernel_primitive:NN \suppressoutererror</code>	<code>\tex_suppressoutererror:D</code>
920	<code>_kernel_primitive:NN \suppressprimitiveerror</code>	
921	<code>\tex_suppressprimitiveerror:D</code>	
922	<code>_kernel_primitive:NN \textdir</code>	<code>\tex_textdir:D</code>
923	<code>_kernel_primitive:NN \textdirection</code>	<code>\tex_textdirection:D</code>
924	<code>_kernel_primitive:NN \toksapp</code>	<code>\tex_toksapp:D</code>
925	<code>_kernel_primitive:NN \tokspre</code>	<code>\tex_tokspre:D</code>
926	<code>_kernel_primitive:NN \tpack</code>	<code>\tex_tpack:D</code>
927	<code>_kernel_primitive:NN \variablefam</code>	<code>\tex_variablefam:D</code>
928	<code>_kernel_primitive:NN \vpack</code>	<code>\tex_vpack:D</code>
929	<code>_kernel_primitive:NN \wordboundary</code>	<code>\tex_wordboundary:D</code>
930	<code>_kernel_primitive:NN \xtoksapp</code>	<code>\tex_xtoksapp:D</code>
931	<code>_kernel_primitive:NN \xtokspre</code>	<code>\tex_xtokspre:D</code>

Primitives from pdfTeX that LuaTeX renames.

932	<code>_kernel_primitive:NN \adjustspacing</code>	<code>\tex_adjustspacing:D</code>
933	<code>_kernel_primitive:NN \copyfont</code>	<code>\tex_copyfont:D</code>
934	<code>_kernel_primitive:NN \draftmode</code>	<code>\tex_draftmode:D</code>

935	_kernel_primitive:NN	\expandglyphsinfont	\tex_fontexpand:D
936	_kernel_primitive:NN	\ifabsdim	\tex_ifabsdim:D
937	_kernel_primitive:NN	\ifabsnum	\tex_ifabsnum:D
938	_kernel_primitive:NN	\ignoreligaturesinfont	\tex_ignoreligaturesinfont:D
939	_kernel_primitive:NN	\insertht	\tex_insertht:D
940	_kernel_primitive:NN	\lastsavedboxresourceindex	
941		\tex_pdflastxform:D	
942	_kernel_primitive:NN	\lastsavedimageresourceindex	
943		\tex_pdflastximage:D	
944	_kernel_primitive:NN	\lastsavedimageresourcepages	
945		\tex_pdflastximagepages:D	
946	_kernel_primitive:NN	\lastxpos	\tex_lastxpos:D
947	_kernel_primitive:NN	\lastypos	\tex_lastypos:D
948	_kernel_primitive:NN	\normaldeviate	\tex_normaldeviate:D
949	_kernel_primitive:NN	\outputmode	\tex_pdfoutput:D
950	_kernel_primitive:NN	\pageheight	\tex_pageheight:D
951	_kernel_primitive:NN	\pagewidth	\tex_pagewidth:D
952	_kernel_primitive:NN	\protrudechars	\tex_protrudechars:D
953	_kernel_primitive:NN	\pxdimen	\tex_pxdimen:D
954	_kernel_primitive:NN	\randomseed	\tex_randomseed:D
955	_kernel_primitive:NN	\useboxresource	\tex_pdfrefxform:D
956	_kernel_primitive:NN	\useimageresource	\tex_pdfrefximage:D
957	_kernel_primitive:NN	\savepos	\tex_savepos:D
958	_kernel_primitive:NN	\saveboxresource	\tex_pdfxform:D
959	_kernel_primitive:NN	\saveimageresource	\tex_pdfximage:D
960	_kernel_primitive:NN	\setrandomseed	\tex_setrandomseed:D
961	_kernel_primitive:NN	\tracingfonts	\tex_tracingfonts:D
962	_kernel_primitive:NN	\uniformdeviate	\tex_uniformdeviate:D

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath....

963	_kernel_primitive:NN	\Uchar	\tex_Uchar:D
964	_kernel_primitive:NN	\Ucharcat	\tex_Ucharcat:D
965	_kernel_primitive:NN	\Udelcode	\tex_Udelcode:D
966	_kernel_primitive:NN	\Udelcodenum	\tex_Udelcodenum:D
967	_kernel_primitive:NN	\Udelimiter	\tex_Udelimiter:D
968	_kernel_primitive:NN	\Udelimiterover	\tex_Udelimiterover:D
969	_kernel_primitive:NN	\Udelimiterunder	\tex_Udelimiterunder:D
970	_kernel_primitive:NN	\Uhextensible	\tex_Uhextensible:D
971	_kernel_primitive:NN	\Uleft	\tex_Uleft:D
972	_kernel_primitive:NN	\Umathaccent	\tex_Umathaccent:D
973	_kernel_primitive:NN	\Umathaxis	\tex_Umathaxis:D
974	_kernel_primitive:NN	\Umathbinbinspacing	\tex_Umathbinbinspacing:D
975	_kernel_primitive:NN	\Umathbinclosespacing	\tex_Umathbinclosespacing:D
976	_kernel_primitive:NN	\Umathbininnerspacing	\tex_Umathbininnerspacing:D
977	_kernel_primitive:NN	\Umathbinopenspacing	\tex_Umathbinopenspacing:D
978	_kernel_primitive:NN	\Umathbinopspacing	\tex_Umathbinopspacing:D
979	_kernel_primitive:NN	\Umathbinordspacing	\tex_Umathbinordspacing:D
980	_kernel_primitive:NN	\Umathbinpunctspacing	\tex_Umathbinpunctspacing:D
981	_kernel_primitive:NN	\Umathbinrelspacing	\tex_Umathbinrelspacing:D
982	_kernel_primitive:NN	\Umathchar	\tex_Umathchar:D
983	_kernel_primitive:NN	\Umathcharclass	\tex_Umathcharclass:D
984	_kernel_primitive:NN	\Umathchardef	\tex_Umathchardef:D

```

985 \__kernel_primitive:NN \Umathcharfam \tex_Umathcharfam:D
986 \__kernel_primitive:NN \Umathcharnum \tex_Umathcharnum:D
987 \__kernel_primitive:NN \Umathcharnumdef \tex_Umathcharnumdef:D
988 \__kernel_primitive:NN \Umathcharslot \tex_Umathcharslot:D
989 \__kernel_primitive:NN \Umathclosebinspacing \tex_Umathclosebinspacing:D
990 \__kernel_primitive:NN \Umathcloseclosespacing
991 \tex_Umathcloseclosespacing:D
992 \__kernel_primitive:NN \Umathcloseinnerspacing
993 \tex_Umathcloseinnerspacing:D
994 \__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
995 \__kernel_primitive:NN \Umathcloseopspacing \tex_Umathcloseopspacing:D
996 \__kernel_primitive:NN \Umathcloseordspacing \tex_Umathcloseordspacing:D
997 \__kernel_primitive:NN \Umathclosepunctspacing
998 \tex_Umathclosepunctspacing:D
999 \__kernel_primitive:NN \Umathcloserelspacing \tex_Umathcloserelspacing:D
1000 \__kernel_primitive:NN \Umathcode \tex_Umathcode:D
1001 \__kernel_primitive:NN \Umathcodenum \tex_Umathcodenum:D
1002 \__kernel_primitive:NN \Umathconnectoroverlapmin
1003 \tex_Umathconnectoroverlapmin:D
1004 \__kernel_primitive:NN \Umathfractiondelsize \tex_Umathfractiondelsize:D
1005 \__kernel_primitive:NN \Umathfractiondenomdown
1006 \tex_Umathfractiondenomdown:D
1007 \__kernel_primitive:NN \Umathfractiondenomvgap
1008 \tex_Umathfractiondenomvgap:D
1009 \__kernel_primitive:NN \Umathfractionnumup \tex_Umathfractionnumup:D
1010 \__kernel_primitive:NN \Umathfractionnumvgap \tex_Umathfractionnumvgap:D
1011 \__kernel_primitive:NN \Umathfractionrule \tex_Umathfractionrule:D
1012 \__kernel_primitive:NN \Umathinnerbinspacing \tex_Umathinnerbinspacing:D
1013 \__kernel_primitive:NN \Umathinnerclosespacing
1014 \tex_Umathinnerclosespacing:D
1015 \__kernel_primitive:NN \Umathinnerinnerspacing
1016 \tex_Umathinnerinnerspacing:D
1017 \__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
1018 \__kernel_primitive:NN \Umathinneropspacing \tex_Umathinneropspacing:D
1019 \__kernel_primitive:NN \Umathinnerordspacing \tex_Umathinnerordspacing:D
1020 \__kernel_primitive:NN \Umathinnerpunctspacing
1021 \tex_Umathinnerpunctspacing:D
1022 \__kernel_primitive:NN \Umathinnerrelspacing \tex_Umathinnerrelspacing:D
1023 \__kernel_primitive:NN \Umathlimitabovebgap \tex_Umathlimitabovebgap:D
1024 \__kernel_primitive:NN \Umathlimitabovekern \tex_Umathlimitabovekern:D
1025 \__kernel_primitive:NN \Umathlimitabovevgap \tex_Umathlimitabovevgap:D
1026 \__kernel_primitive:NN \Umathlimitbelowbgap \tex_Umathlimitbelowbgap:D
1027 \__kernel_primitive:NN \Umathlimitbelowkern \tex_Umathlimitbelowkern:D
1028 \__kernel_primitive:NN \Umathlimitbelowvgap \tex_Umathlimitbelowvgap:D
1029 \__kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
1030 \__kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
1031 \__kernel_primitive:NN \Umathopbinspacing \tex_Umathopbinspacing:D
1032 \__kernel_primitive:NN \Umathopclosespacing \tex_Umathopclosespacing:D
1033 \__kernel_primitive:NN \Umathopenbinspacing \tex_Umathopenbinspacing:D
1034 \__kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D
1035 \__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1036 \__kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1037 \__kernel_primitive:NN \Umathopenopspacing \tex_Umathopenopspacing:D
1038 \__kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D

```

1039 __kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1040 __kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1041 __kernel_primitive:NN \Umathoperatorsize \tex_Umathoperatorsize:D
1042 __kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D
1043 __kernel_primitive:NN \Umathopopenspacing \tex_Umathopopenspacing:D
1044 __kernel_primitive:NN \Umathopopspacing \tex_Umathopopspacing:D
1045 __kernel_primitive:NN \Umathopordspacing \tex_Umathopordspacing:D
1046 __kernel_primitive:NN \Umathoppunctspacing \tex_Umathoppunctspacing:D
1047 __kernel_primitive:NN \Umathoprelspacing \tex_Umathoprelspacing:D
1048 __kernel_primitive:NN \Umathordbinspacing \tex_Umathordbinspacing:D
1049 __kernel_primitive:NN \Umathordclosespacing \tex_Umathordclosespacing:D
1050 __kernel_primitive:NN \Umathordinnerspacing \tex_Umathordinnerspacing:D
1051 __kernel_primitive:NN \Umathordopenspacing \tex_Umathordopenspacing:D
1052 __kernel_primitive:NN \Umathordopspacing \tex_Umathordopspacing:D
1053 __kernel_primitive:NN \Umathordordspacing \tex_Umathordordspacing:D
1054 __kernel_primitive:NN \Umathordpunctspacing \tex_Umathordpunctspacing:D
1055 __kernel_primitive:NN \Umathordrelspacing \tex_Umathordrelspacing:D
1056 __kernel_primitive:NN \Umathoverbarkern \tex_Umathoverbarkern:D
1057 __kernel_primitive:NN \Umathoverbarrule \tex_Umathoverbarrule:D
1058 __kernel_primitive:NN \Umathoverbarvgap \tex_Umathoverbarvgap:D
1059 __kernel_primitive:NN \Umathoverdelimiterbgap
1060 \tex_Umathoverdelimiterbgap:D
1061 __kernel_primitive:NN \Umathoverdelimitervgap
1062 \tex_Umathoverdelimitervgap:D
1063 __kernel_primitive:NN \Umathpunctbinspacing \tex_Umathpunctbinspacing:D
1064 __kernel_primitive:NN \Umathpunctclosespacing
1065 \tex_Umathpunctclosespacing:D
1066 __kernel_primitive:NN \Umathpunctinnerspacing
1067 \tex_Umathpunctinnerspacing:D
1068 __kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
1069 __kernel_primitive:NN \Umathpunctopspacing \tex_Umathpunctopspacing:D
1070 __kernel_primitive:NN \Umathpunctordspacing \tex_Umathpunctordspacing:D
1071 __kernel_primitive:NN \Umathpunctpunctspacing
1072 \tex_Umathpunctpunctspacing:D
1073 __kernel_primitive:NN \Umathpunctrelspacing \tex_Umathpunctrelspacing:D
1074 __kernel_primitive:NN \Umathquad \tex_Umathquad:D
1075 __kernel_primitive:NN \Umathradicaldegreeafter
1076 \tex_Umathradicaldegreeafter:D
1077 __kernel_primitive:NN \Umathradicaldegreebefore
1078 \tex_Umathradicaldegreebefore:D
1079 __kernel_primitive:NN \Umathradicaldegreeraise
1080 \tex_Umathradicaldegreeraise:D
1081 __kernel_primitive:NN \Umathradicalkern \tex_Umathradicalkern:D
1082 __kernel_primitive:NN \Umathradicalrule \tex_Umathradicalrule:D
1083 __kernel_primitive:NN \Umathradicalvgap \tex_Umathradicalvgap:D
1084 __kernel_primitive:NN \Umathrelbinspacing \tex_Umathrelbinspacing:D
1085 __kernel_primitive:NN \Umathrelclosespacing \tex_Umathrelclosespacing:D
1086 __kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1087 __kernel_primitive:NN \Umathrelopenspacing \tex_Umathrelopenspacing:D
1088 __kernel_primitive:NN \Umathrelopspacing \tex_Umathrelopspacing:D
1089 __kernel_primitive:NN \Umathrelordspacing \tex_Umathrelordspacing:D
1090 __kernel_primitive:NN \Umathrelpunctspacing \tex_Umathrelpunctspacing:D
1091 __kernel_primitive:NN \Umathrelrelspacing \tex_Umathrelrelspacing:D
1092 __kernel_primitive:NN \Umathskewedfractionhgap

1093	\tex_Umathskewedfractionhgap:D	
1094	__kernel_primitive:NN \Umathskewedfractionvgap	
1095	\tex_Umathskewedfractionvgap:D	
1096	__kernel_primitive:NN \Umathspaceafterscript	\tex_Umathspaceafterscript:D
1097	__kernel_primitive:NN \Umathstackdenomdown	\tex_Umathstackdenomdown:D
1098	__kernel_primitive:NN \Umathstacknumup	\tex_Umathstacknumup:D
1099	__kernel_primitive:NN \Umathstackvgap	\tex_Umathstackvgap:D
1100	__kernel_primitive:NN \Umathsubshiftdown	\tex_Umathsubshiftdown:D
1101	__kernel_primitive:NN \Umathsubshiftdrop	\tex_Umathsubshiftdrop:D
1102	__kernel_primitive:NN \Umathsubsupshiftdown	\tex_Umathsubsupshiftdown:D
1103	__kernel_primitive:NN \Umathsubsupvgap	\tex_Umathsubsupvgap:D
1104	__kernel_primitive:NN \Umathsubtopmax	\tex_Umathsubtopmax:D
1105	__kernel_primitive:NN \Umathsupbottommin	\tex_Umathsupbottommin:D
1106	__kernel_primitive:NN \Umathsupshiftdrop	\tex_Umathsupshiftdrop:D
1107	__kernel_primitive:NN \Umathsupshiftup	\tex_Umathsupshiftup:D
1108	__kernel_primitive:NN \Umathsupsubbottommax	\tex_Umathsupsubbottommax:D
1109	__kernel_primitive:NN \Umathunderbarkern	\tex_Umathunderbarkern:D
1110	__kernel_primitive:NN \Umathunderbarrule	\tex_Umathunderbarrule:D
1111	__kernel_primitive:NN \Umathunderbarvgap	\tex_Umathunderbarvgap:D
1112	__kernel_primitive:NN \Umathunderdelimiterbgap	
1113	\tex_Umathunderdelimiterbgap:D	
1114	__kernel_primitive:NN \Umathunderdelimitervgap	
1115	\tex_Umathunderdelimitervgap:D	
1116	__kernel_primitive:NN \Umiddle	\tex_Umiddle:D
1117	__kernel_primitive:NN \Unosubscript	\tex_Unosubscript:D
1118	__kernel_primitive:NN \Unosuperscript	\tex_Unosuperscript:D
1119	__kernel_primitive:NN \Uoverdelimiter	\tex_Uoverdelimiter:D
1120	__kernel_primitive:NN \Uradical	\tex_Uradical:D
1121	__kernel_primitive:NN \Uright	\tex_Uright:D
1122	__kernel_primitive:NN \Uroot	\tex_Uroot:D
1123	__kernel_primitive:NN \Uskewed	\tex_Uskewed:D
1124	__kernel_primitive:NN \Uskewedwithdelims	\tex_Uskewedwithdelims:D
1125	__kernel_primitive:NN \Ustack	\tex_Ustack:D
1126	__kernel_primitive:NN \Ustartdisplaymath	\tex_Ustartdisplaymath:D
1127	__kernel_primitive:NN \Ustartmath	\tex_Ustartmath:D
1128	__kernel_primitive:NN \Ustopdisplaymath	\tex_Ustopdisplaymath:D
1129	__kernel_primitive:NN \Ustopmath	\tex_Ustopmath:D
1130	__kernel_primitive:NN \Usubscript	\tex_Usubscript:D
1131	__kernel_primitive:NN \Usuperscript	\tex_Usuperscript:D
1132	__kernel_primitive:NN \Uunderdelimiter	\tex_Uunderdelimiter:D
1133	__kernel_primitive:NN \Uvextensible	\tex_Uvextensible:D

Primitives from pTeX.

1134	__kernel_primitive:NN \autospaceing	\tex_autospaceing:D
1135	__kernel_primitive:NN \autoxspaceing	\tex_autoxspaceing:D
1136	__kernel_primitive:NN \currentcjktoken	\tex_currentcjktoken:D
1137	__kernel_primitive:NN \currentspacingmode	\tex_currentspacingmode:D
1138	__kernel_primitive:NN \currentxspacingmode	\tex_currentxspacingmode:D
1139	__kernel_primitive:NN \disinhibitglue	\tex_disinhibitglue:D
1140	__kernel_primitive:NN \dtou	\tex_dtou:D
1141	__kernel_primitive:NN \epTeXinputencoding	\tex_epTeXinputencoding:D
1142	__kernel_primitive:NN \epTeXversion	\tex_epTeXversion:D
1143	__kernel_primitive:NN \euc	\tex_euc:D
1144	__kernel_primitive:NN \hfi	\tex_hfi:D
1145	__kernel_primitive:NN \ifdbbox	\tex_ifdbbox:D

1146	_kernel_primitive:NN	\ifddir	\tex_ifddir:D
1147	_kernel_primitive:NN	\ifjfont	\tex_ifjfont:D
1148	_kernel_primitive:NN	\ifmbox	\tex_ifmbox:D
1149	_kernel_primitive:NN	\ifmdir	\tex_ifmdir:D
1150	_kernel_primitive:NN	\iftbox	\tex_iftbox:D
1151	_kernel_primitive:NN	\iftfont	\tex_iftfont:D
1152	_kernel_primitive:NN	\iftdir	\tex_iftdir:D
1153	_kernel_primitive:NN	\ifybox	\tex_ifybox:D
1154	_kernel_primitive:NN	\ifydir	\tex_ifydir:D
1155	_kernel_primitive:NN	\inhibitglue	\tex_inhibitglue:D
1156	_kernel_primitive:NN	\inhibitxspcode	\tex_inhibitxspcode:D
1157	_kernel_primitive:NN	\jcharwidowpenalty	\tex_jcharwidowpenalty:D
1158	_kernel_primitive:NN	\jfam	\tex_jfam:D
1159	_kernel_primitive:NN	\jfont	\tex_jfont:D
1160	_kernel_primitive:NN	\jis	\tex_jis:D
1161	_kernel_primitive:NN	\kanjiskip	\tex_kanjiskip:D
1162	_kernel_primitive:NN	\kansuji	\tex_kansuji:D
1163	_kernel_primitive:NN	\kansujichar	\tex_kansujichar:D
1164	_kernel_primitive:NN	\kcatcode	\tex_kcatcode:D
1165	_kernel_primitive:NN	\kuten	\tex_kuten:D
1166	_kernel_primitive:NN	\lastnodechar	\tex_lastnodechar:D
1167	_kernel_primitive:NN	\lastnodefont	\tex_lastnodefont:D
1168	_kernel_primitive:NN	\lastnodesubtype	\tex_lastnodesubtype:D
1169	_kernel_primitive:NN	\noautospaceing	\tex_noautospaceing:D
1170	_kernel_primitive:NN	\noautoxspaceing	\tex_noautoxspaceing:D
1171	_kernel_primitive:NN	\pagefistretch	\tex_pagefistretch:D
1172	_kernel_primitive:NN	\postbreakpenalty	\tex_postbreakpenalty:D
1173	_kernel_primitive:NN	\prebreakpenalty	\tex_prebreakpenalty:D
1174	_kernel_primitive:NN	\ptexfontname	\tex_ptexfontname:D
1175	_kernel_primitive:NN	\ptexlineendmode	\tex_lineendmode:D
1176	_kernel_primitive:NN	\ptexminorversion	\tex_ptexminorversion:D
1177	_kernel_primitive:NN	\ptexrevision	\tex_ptexrevision:D
1178	_kernel_primitive:NN	\ptextracingfonts	\tex_ptextracingfonts:D
1179	_kernel_primitive:NN	\ptexversion	\tex_ptexversion:D
1180	_kernel_primitive:NN	\readpapersizespecial	\tex_readpapersizespecial:D
1181	_kernel_primitive:NN	\scriptbaselineshiftfactor	
1182		\tex_scriptbaselineshiftfactor:D	
1183	_kernel_primitive:NN	\scriptscriptbaselineshiftfactor	
1184		\tex_scriptscriptbaselineshiftfactor:D	
1185	_kernel_primitive:NN	\showmode	\tex_showmode:D
1186	_kernel_primitive:NN	\sjis	\tex_sjis:D
1187	_kernel_primitive:NN	\tate	\tex_tate:D
1188	_kernel_primitive:NN	\tbaselineshift	\tex_tbaselineshift:D
1189	_kernel_primitive:NN	\textbaselineshiftfactor	
1190		\tex_textbaselineshiftfactor:D	
1191	_kernel_primitive:NN	\tfont	\tex_tfont:D
1192	_kernel_primitive:NN	\tojis	\tex_tojis:D
1193	_kernel_primitive:NN	\toucs	\tex_toucs:D
1194	_kernel_primitive:NN	\ucs	\tex_ucs:D
1195	_kernel_primitive:NN	\xkanjiskip	\tex_xkanjiskip:D
1196	_kernel_primitive:NN	\xspcode	\tex_xspcode:D
1197	_kernel_primitive:NN	\ybaselineshift	\tex_ybaselineshift:D
1198	_kernel_primitive:NN	\yoko	\tex_yoko:D
1199	_kernel_primitive:NN	\vfi	\tex_vfi:D

Primitives from upTeX.

1200	<code>__kernel_primitive:NN \disablecjktoken</code>	<code>\tex_disablecjktoken:D</code>
1201	<code>__kernel_primitive:NN \enablecjktoken</code>	<code>\tex_enablecjktoken:D</code>
1202	<code>__kernel_primitive:NN \forcecjktoken</code>	<code>\tex_forcecjktoken:D</code>
1203	<code>__kernel_primitive:NN \kchar</code>	<code>\tex_kchar:D</code>
1204	<code>__kernel_primitive:NN \kchardef</code>	<code>\tex_kchardef:D</code>
1205	<code>__kernel_primitive:NN \uptexrevision</code>	<code>\tex_uptexrevision:D</code>
1206	<code>__kernel_primitive:NN \uptexversion</code>	<code>\tex_uptexversion:D</code>

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

1207	<code>__kernel_primitive:NN \odelcode</code>	<code>\tex_odelcode:D</code>
1208	<code>__kernel_primitive:NN \odelimiter</code>	<code>\tex_odelimiter:D</code>
1209	<code>__kernel_primitive:NN \omathaccent</code>	<code>\tex_omathaccent:D</code>
1210	<code>__kernel_primitive:NN \omathchar</code>	<code>\tex_omathchar:D</code>
1211	<code>__kernel_primitive:NN \omathchardef</code>	<code>\tex_omathchardef:D</code>
1212	<code>__kernel_primitive:NN \omathcode</code>	<code>\tex_omathcode:D</code>
1213	<code>__kernel_primitive:NN \oradical</code>	<code>\tex_oradical:D</code>

Newer cross-engine primitives.

1214	<code>__kernel_primitive:NN \ignoreprimitiveerror</code>	<code>\tex_ignoreprimitiveerror:D</code>
1215	<code>__kernel_primitive:NN \partokencontext</code>	<code>\tex_partokencontext:D</code>
1216	<code>__kernel_primitive:NN \partokenname</code>	<code>\tex_partokenname:D</code>
1217	<code>__kernel_primitive:NN \showstream</code>	<code>\tex_showstream:D</code>
1218	<code>__kernel_primitive:NN \tracingstacklevels</code>	<code>\tex_tracingstacklevels:D</code>

End of the “just the names” part of the source.

```
1219 </names | code>
1220 <*code>
```

The job is done: close the group (using the primitive renamed!).

```
1221 \tex_endgroup:D
```

L^AT_EX 2_ε moves a few primitives, so these are sorted out. In newer versions of L^AT_EX 2_ε, expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the L^AT_EX 2_ε format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```
1222 \tex_ifdefined:D \@@end
1223 \tex_let:D \tex_end:D \@@end
1224 \tex_let:D \tex_input:D \@@input
1225 \tex_fi:D
```

If `\@@@hyph` is defined, we are loading expl3 in a pre-2020/10/01 release of L^AT_EX 2_ε, so a few other primitives have to be tested as well.

```
1226 \tex_ifdefined:D \@@hyph
1227 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1228 \tex_let:D \tex_everymath:D \frozen@everymath
1229 \tex_let:D \tex_hyphen:D \@@hyph
1230 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1231 \tex_let:D \tex_underline:D \@@underline
```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument`

hook. (We cannot use `\primitive` as that doesn't allow us to make a direct copy of the primitive *itself*.) As we know that L^AT_EX 2_ε is in use, we use its `\@tfor` loop here.

```

1232 \tex_ifdefined:D \@@shipout
1233 \tex_let:D \tex_shipout:D \@@shipout
1234 \tex_fi:D
1235 \tex_begingroup:D
1236 \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
1237 \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
1238 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1239 \tex_else:D
1240 \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
1241 \CROP@shipout
1242 \dup@shipout
1243 \GPTorg@shipout
1244 \LL@shipout
1245 \mem@oldshipout
1246 \opem@shipout
1247 \pgfpages@originalshipout
1248 \pr@shipout
1249 \Shipout
1250 \verso@orig@shipout
1251 \do
1252 {
1253 \tex_edef:D \l_tmpb_tl
1254 { \tex_expandafter:D \tex_meaning:D \@tempa }
1255 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1256 \tex_global:D \tex_expandafter:D \tex_let:D
1257 \tex_expandafter:D \tex_shipout:D \@tempa
1258 \tex_fi:D
1259 }
1260 \tex_fi:D
1261 \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer L^AT_EX has this simply as `\tracingfonts`, but that is overwritten by the L^AT_EX 2_ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfT_EX name or from L^AT_EX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L^AT_EX 2_ε users will have `expl3` loaded by `fontspec`. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1262 \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1263 \tex_ifdefined:D \pdftracingfonts
1264 \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1265 \tex_else:D
1266 \tex_ifdefined:D \tex_directlua:D
1267 \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1268 \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1269 \tex_fi:D
1270 \tex_fi:D
1271 \tex_fi:D

```

Only pdfT_EX and L^AT_EX define `\pdfmapfile` and `\pdfmapline`: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1272 \tex_ifnum:D 0
1273 \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D

```

```

1274 \tex_ifdefined:D \tex_luatexversion:D 1 \tex_fi:D
1275 = 0 %
1276 \tex_let:D \tex_pdfmapfile:D \tex_undefined:D
1277 \tex_let:D \tex_pdfmapline:D \tex_undefined:D
1278 \tex_fi:D

```

A few packages do unfortunate things to date-related primitives.

```

1279 \tex_begingroup:D
1280 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_time:D }
1281 \tex_edef:D \l_tmpb_tl { \tex_string:D \time }
1282 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1283 \tex_else:D
1284 \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1285 \tex_fi:D
1286 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_day:D }
1287 \tex_edef:D \l_tmpb_tl { \tex_string:D \day }
1288 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1289 \tex_else:D
1290 \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1291 \tex_fi:D
1292 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_month:D }
1293 \tex_edef:D \l_tmpb_tl { \tex_string:D \month }
1294 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1295 \tex_else:D
1296 \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1297 \tex_fi:D
1298 \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_year:D }
1299 \tex_edef:D \l_tmpb_tl { \tex_string:D \year }
1300 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1301 \tex_else:D
1302 \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1303 \tex_fi:D
1304 \tex_endgroup:D

```

cs_lat_{ex} moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```

1305 \tex_ifdefined:D \orieveryjob
1306 \tex_let:D \tex_everyjob:D \orieveryjob
1307 \tex_fi:D
1308 \tex_ifdefined:D \oripdfoutput
1309 \tex_let:D \tex_pdfoutput:D \oripdfoutput
1310 \tex_fi:D

```

For ConT_EX_t, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConT_EX_t.

```

1311 \tex_ifdefined:D \normalend
1312 \tex_let:D \tex_end:D \normalend
1313 \tex_let:D \tex_everyjob:D \normaleveryjob
1314 \tex_let:D \tex_input:D \normalinput
1315 \tex_let:D \tex_language:D \normallanguage
1316 \tex_let:D \tex_mathop:D \normalmathop
1317 \tex_let:D \tex_month:D \normalmonth
1318 \tex_let:D \tex_outer:D \normalouter

```

```

1319 \tex_let:D \tex_over:D \normalover
1320 \tex_let:D \tex_vcenter:D \normalvcenter
1321 \tex_let:D \tex_unexpanded:D \normalunexpanded
1322 \tex_let:D \tex_expanded:D \normalexpanded
1323 \tex_fi:D
1324 \tex_ifdefined:D \normalitaliccorrection
1325 \tex_let:D \tex_hoffset:D \normalhoffset
1326 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1327 \tex_let:D \tex_voffset:D \normalvoffset
1328 \tex_let:D \tex_showtokens:D \normalshowtokens
1329 \tex_fi:D
1330 \tex_ifdefined:D \normalleft
1331 \tex_let:D \tex_left:D \normalleft
1332 \tex_let:D \tex_middle:D \normalmiddle
1333 \tex_let:D \tex_right:D \normalright
1334 \tex_fi:D
1335 \code

```

In LuaTeX, we additionally emulate some primitives using Lua code.

```

1336 \lua

```

`\tex_strcmp:D` Compare two strings, expanding to 0 if they are equal, -1 if the first one is smaller and 1 if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```

1337 local minus_tok = token_new(string.byte'-', 12)
1338 local zero_tok = token_new(string.byte'0', 12)
1339 local one_tok = token_new(string.byte'1', 12)
1340 luacmd('tex_strcmp:D', function()
1341   local first = scan_string()
1342   local second = scan_string()
1343   if first < second then
1344     put_next(minus_tok, one_tok)
1345   else
1346     put_next(first == second and zero_tok or one_tok)
1347   end
1348 end, 'global')

```

(End of definition for \tex_strcmp:D.)

`\tex_Ucharcat:D` Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.new(...)` is about 10% slower but needed to create arbitrary space tokens.

```

1349 local sprint = tex.sprint
1350 local cprint = tex.cprint
1351 luacmd('tex_Ucharcat:D', function()
1352   local charcode = scan_int()
1353   local catcode = scan_int()
1354   if catcode == 10 then
1355     sprint(token_new(charcode, 10))
1356   else
1357     cprint(catcode, utf8_char(charcode))
1358   end
1359 end, 'global')

```

(End of definition for \tex_Ucharcat:D.)

`\tex_filesize:D` Wrap the function from `ltxutils`.

```
1360 luacmd('tex_filesize:D', function()  
1361   local size = filesize(scan_string())  
1362   if size then write(size) end  
1363 end, 'global')
```

(End of definition for \tex_filesize:D.)

`\tex_md5sum:D` There are two cases: Either hash a file or a string. Both are already implemented in `l3luatex` or built-in.

```
1364 luacmd('tex_md5sum:D', function()  
1365   local hash  
1366   if scan_keyword"file" then  
1367     hash = filemd5sum(scan_string())  
1368   else  
1369     hash = md5_HEX(scan_string())  
1370   end  
1371   if hash then write(hash) end  
1372 end, 'global')
```

(End of definition for \tex_md5sum:D.)

`\tex_filemoddate:D` A primitive for getting the modification date of a file.

```
1373 luacmd('tex_filemoddate:D', function()  
1374   local date = filemoddate(scan_string())  
1375   if date then write(date) end  
1376 end, 'global')
```

(End of definition for \tex_filemoddate:D.)

`\tex_filedump:D` An emulated primitive for getting a hexdump from a (partial) file. The length has a default of 0. This is consistent with pdfTeX, but it effectively makes the primitive useless without an explicit `length`. Therefore we allow the keyword `whole` to be used instead of a length, indicating that the whole remaining file should be read.

```
1377 luacmd('tex_filedump:D', function()  
1378   local offset = scan_keyword'offset' and scan_int() or nil  
1379   local length = scan_keyword'length' and scan_int()  
1380                 or not scan_keyword'whole' and 0 or nil  
1381   local data = filedump(scan_string(), offset, length)  
1382   if data then write(data) end  
1383 end, 'global')
```

(End of definition for \tex_filedump:D.)

```
1384 </lua>
```

Chapter 45

l3kernel-functions: kernel-reserved functions

45.1 Internal l3debug kernel functions

These function are only created if debugging is enabled, hence they are actually defined in l3debug.

<code>__kernel_chk_var_local:N</code>	<code>__kernel_chk_var_local:N <var></code>
<code>__kernel_chk_var_global:N</code>	<code>__kernel_chk_var_global:N <var></code>

Applies `\__kernel_chk_var_exist:N <var>` as well as `\__kernel_chk_var_scope:NN <scope> <var>`, where `<scope>` is l or g.

<code>__kernel_chk_var_scope:NN</code>	<code>__kernel_chk_var_scope:NN <scope> <var></code>

Checks the `<var>` has the correct `<scope>`, and if not raises a kernel-level error. The `<scope>` is a single letter l, g, c denoting local variables, global variables, or constants. More precisely, if the variable name starts with a letter and an underscore (normal expl3 convention) the function checks that this single letter matches the `<scope>`. Otherwise the function cannot know the scope `<var>` the first time: instead, it defines `\__debug_chk_/<var name>` to store that information for the next call. Thus, if a given `<var>` is subject to assignments of different scopes a kernel error will result.

<code>__kernel_chk_cs_exist:N</code>	<code>__kernel_chk_cs_exist:N <cs></code>
<code>__kernel_chk_cs_exist:c</code>	<code>__kernel_chk_var_exist:N <var></code>
<code>__kernel_chk_var_exist:N</code>	

Checks that their argument is defined according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error. Error messages are different.

<code>__kernel_chk_flag_exist:NN</code>	<code>__kernel_chk_flag_exist:NN</code>
	<code><function> <flag></code>

Checks that the `<flag>` is defined according to the criterion for `\flag_if_exist_p:N`, and if not raises a kernel-level error and calls the function with the argument `\l_tmpa_flag` to proceed somehow without producing too many errors.

`\__kernel_debug_log:e` `\__kernel_debug_log:e {⟨message text⟩}`

If the `log-functions` option is active, this function writes the `⟨message text⟩` to the log file using `\iow_log:e`. Otherwise, the `⟨message text⟩` is ignored using `\use_none:n`.

45.2 Internal kernel functions

`\__kernel_chk_defined:NT` `\__kernel_chk_defined:NT {⟨variable⟩} {⟨true code⟩}`

If `⟨variable⟩` is not defined (according to `\cs_if_exist:NTF`), this triggers an error, otherwise the `⟨true code⟩` is run.

`\__kernel_chk_expr:nNnN` `\__kernel_chk_expr:nNnN {⟨expr⟩} {⟨eval⟩} {⟨convert⟩} {⟨caller⟩}`

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nnnn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D {⟨eval⟩} {⟨expr⟩} \tex_relax:D` after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the `⟨caller⟩`. For instance `⟨eval⟩` can be `\__int_eval:w` and `⟨caller⟩` can be `\int_eval:n` or `\int_set:Nn`. The argument `⟨convert⟩` is empty except for mu expressions where it is `\tex_mutoglue:D`, used for internal purposes.

`\__kernel_chk_tl_type:NnnT` `\__kernel_chk_tl_type:NnnT {⟨control sequence⟩} {⟨specific type⟩} {⟨reconstruction⟩} {⟨true code⟩}`

Helper to test that the `⟨control sequence⟩` is a variable of the given `⟨specific type⟩` of token list. Produces suitable error messages if the `⟨control sequence⟩` does not exist, or if it is not a token list variable at all, or if the `⟨control sequence⟩` differs from the result of e-expanding `⟨reconstruction⟩`. If all of these tests succeed then the `⟨true code⟩` is run.

`\__kernel_codepoint_to_bytes:n` `\__kernel_codepoint_to_bytes:n {⟨codepoint⟩}`

Converts the `⟨codepoint⟩` to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups #1 and #2 filled and #3 and #4 empty.

`\__kernel_codepoint_to_grapheme_class:n` `\__kernel_codepoint_to_grapheme_class:n {⟨codepoint⟩}`
`\__kernel_codepoint_to_wordbreak_class:n` `\__kernel_codepoint_to_wordbreak_class:n {⟨codepoint⟩}`

Expands to the Unicode properties `Grapheme_Cluster_Break` and `Word_Break`, respectively, of the `⟨codepoint⟩`. Here, `Grapheme_Cluster_Break` and `Word_Break` are strings matching exactly the descriptors given in <https://www.unicode.org/reports/tr29/>.

```
\_kernel_cs_parm_from_arg_count:nnF \_kernel_cs_parm_from_arg_count:nnF {<follow-on>} {<args>} {<>false  
code>}
```

Evaluates the number of `<args>` and leaves the `<follow-on>` code followed by a brace group containing the required number of primitive parameter markers (`#1`, *etc.*). If the number of `<args>` is outside the range `[0,9]`, the `<>false code>` is inserted *instead* of the `<follow-on>`.

```
\_kernel_dependency_version_check:Nn \_kernel_dependency_version_check:Nn {<\date>} {<file>}  
\_kernel_dependency_version_check:nn \_kernel_dependency_version_check:nn {<date>} {<file>}
```

Checks if the loaded version of the expl3 kernel is at least `<date>`, required by `<file>`. If the kernel date is older than `<date>`, the loading of `<file>` is aborted and an error is raised.

```
\_kernel_deprecation_code:nn \_kernel_deprecation_code:nn {<error code>} {<working code>}
```

Stores both an `<error>` and `<working>` definition for given material such that they can be exchanged by `\debug_on:n` and `\debug_off:n`.

```
\_kernel_exp_not:w * \_kernel_exp_not:w <expandable tokens> {<content>}
```

Carries out expansion on the `<expandable tokens>` before preventing further expansion of the `<content>` as for `\exp_not:n`. Typically, the `<expandable tokens>` will alter the nature of the `<content>`, i.e., allow it to be generated in some way.

```
\l__kernel_expl_bool
```

A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn/\ExplSyntaxOff`.

(End of definition for `\l__kernel_expl_bool`.)

```
\c__kernel_expl_date_tl
```

A token list containing the release date of the l3kernel preloaded in L^AT_EX 2_ε used to check if dependencies match.

(End of definition for `\c__kernel_expl_date_tl`.)

```
\_kernel_file_missing:n \_kernel_file_missing:n {<name>}
```

Expands the `<name>` as per `\_kernel_file_name_sanitize:n` then produces an error message indicating that this file was not found.

```
\_kernel_file_name_sanitize:n * \_kernel_file_name_sanitize:n {<name>}
```

Updated: 2021-04-17

Expands the file name using a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

```
\_kernel_file_input_push:n \_kernel_file_input_push:n {<name>}  
\_kernel_file_input_pop: \_kernel_file_input_pop:
```

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L^AT_EX 2_ε kernel is necessary.

`\_kernel\_int\_add:nnn` ★ `\_kernel\_int\_add:nnn {⟨integer1⟩} {⟨integer2⟩} {⟨integer3⟩}`

Expands to the result of adding the three *⟨integer_s⟩* (which must be suitable input for `\int\_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31} + 1, 2^{31} - 1]$. The *⟨integer_s⟩* may be of the form `\int\_eval:w ... \scan\_stop:` but may be evaluated more than once.

`\_kernel\_int\_sep:` ★ `\_kernel\_int\_sep:`

Provides a way to stop the expansion in `\tex\_numexpr:D` with a token that is not absorbed so that it works as a marker for right-delimited arguments. Additionally the used token should be impossible to form a valid expression for these expression primitives (the otherwise previously-used `;` is turned into a binary operator in some engines for instance). An unexpandable primitive is used because it is guaranteed to never be a valid token in such an expression and survives all kinds of expansion (just like `;` would in ε -TeX).

`\_kernel\_intarray\_gset:Nnn` `\_kernel\_intarray\_gset:Nnn ⟨intarray var⟩ {⟨index⟩} {⟨value⟩}`

Faster version of `\intarray\_gset:Nnn`. Stores the *⟨value⟩* into the *⟨integer array variable⟩* at the *⟨position⟩*. The *⟨index⟩* and *⟨value⟩* must be suitable for a direct assignment to a TeX count register, for instance expanding to an integer denotation or obtained through the primitive `\numexpr` (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the *⟨position⟩* is between 1 and the `\intarray\_count:N`, and the *⟨value⟩*'s absolute value is at most $2^{30} - 1$. Assignments are always global.

`\_kernel\_intarray\_item:Nn` ★ `\_kernel\_intarray\_item:Nn ⟨intarray var⟩ {⟨index⟩}`

Faster version of `\intarray\_item:Nn`. Expands to the integer entry stored at the *⟨index⟩* in the *⟨integer array variable⟩*. The *⟨index⟩* must be suitable for a direct assignment to a TeX count register and must be between 1 and the `\intarray\_count:N`, lest a low-level TeX error occur.

`\_kernel\_intarray\_range\_to\_clist:Nnn` ☆ `\_kernel\_intarray\_range\_to\_clist:Nnn ⟨intarray var⟩ {⟨start index⟩} {⟨end index⟩}`

New: 2020-07-12

Converts to integer denotations separated by commas the entries of the *⟨intarray⟩* from positions *⟨start index⟩* to *⟨end index⟩* included. The *⟨start index⟩* and *⟨end index⟩* must be suitable for a direct assignment to a TeX count register, must be between 1 and the `\intarray\_count:N`, and be suitably ordered. All tokens have category code other.

`\_kernel\_intarray\_gset\_range\_from\_clist:Nnn` `\_kernel\_intarray\_gset\_range\_from\_clist:Nnn ⟨intarray var⟩ {⟨start index⟩} {⟨integer clist⟩}`

New: 2020-07-12

Stores the entries of the *⟨clist⟩* as entries of the *⟨intarray var⟩* starting from the *⟨start index⟩*, upwards. This is done without any bound checking. The *⟨start index⟩* and all entries of the *⟨integer comma list⟩* (which do not undergo space trimming and brace stripping as in normal clist mappings) must be suitable for a direct assignment to a TeX count register. An empty entry may stop the loop.

<u>_kernel_ior_open:Nn</u>	<u>_kernel_ior_open:Nn</u> $\langle stream \rangle$ $\{\langle file\ name \rangle\}$
<u>_kernel_ior_open:No</u>	This function has identical syntax to the public version. However, is does not take precautions against active characters in the $\langle file\ name \rangle$, and it does not attempt to add a $\langle path \rangle$ to the $\langle file\ name \rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name \rangle$ and which need to perform multiple open or close operations. See for example the implementation of <code>\ior_shell_open:Nn</code> .
<u>_kernel_iow_open:Nn</u>	<u>_kernel_iow_open:Nn</u> $\langle stream \rangle$ $\{\langle file\ name \rangle\}$
<u>_kernel_iow_open:No</u>	This function has identical syntax to the public version. However, is does not take precautions against active characters in the $\langle file\ name \rangle$, and it does not attempt to add a $\langle path \rangle$ to the $\langle file\ name \rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name \rangle$ and which need to perform multiple open or close operations. See for example the implementation of <code>\iow_shell_open:Nn</code> .
<u>_kernel_iow_with:Nnn</u>	<u>_kernel_iow_with:Nnn</u> $\langle integer \rangle$ $\{\langle value \rangle\}$ $\{\langle code \rangle\}$
	If the $\langle integer \rangle$ is equal to the $\langle value \rangle$ then this function simply runs the $\langle code \rangle$. Otherwise it saves the current value of the $\langle integer \rangle$, sets it to the $\langle value \rangle$, runs the $\langle code \rangle$, and restores the $\langle integer \rangle$ to its former value. This is used to ensure that the <code>\newlinechar</code> is 10 when writing to a stream, which lets <code>\iow_newline:</code> work, and that <code>\errorcontextlines</code> is -1 when displaying a message.
<u>_kernel_msg_show_eval:Nn</u>	<u>_kernel_msg_show_eval:Nn</u> $\langle function \rangle$ $\{\langle expression \rangle\}$
<u>_kernel_msg_log_eval:Nn</u>	Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{\langle expression \rangle\}$ (with <code>f</code> -expansion). For instance, if the $\langle function \rangle$ is <code>\int_eval:n</code> and the $\langle expression \rangle$ is <code>1+2</code> then this logs <code>> 1+2=3</code> .
<u>_kernel_pdf_object_id:n</u>	<u>_kernel_pdf_object_id:n</u> $\star$ <u>_kernel_pdf_object_id:n</u> $\{\langle object \rangle\}$
<u>_kernel_pdf_object_id_indexed:nn</u>	<u>_kernel_pdf_object_id_indexed:nn</u> $\star$ <u>_kernel_pdf_object_id_indexed:nn</u> $\{\langle class \rangle\}$ $\{\langle number \rangle\}$
	Expands to the ID of $\langle object \rangle$ (or object of $\langle number \rangle$ within the $\langle class \rangle$), in for example page resource allocation. Depending on the backend, the result may be the same as <code>\pdf_object_id:n/\pdf_object_id_indexed:nn</code> .
<u>\g__kernel_prg_map_int</u>	This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions <code>\<type>_map_1:w</code> , <code>\<type>_map_2:w</code> , etc., labeled by <code>\g__kernel_prg_map_int</code> hold functions to be mapped over various list datatypes in inline and variable mappings.
	(End of definition for <code>\g__kernel_prg_map_int</code> .)

`\__kernel_quark_new_test:N \__kernel_quark_new_test:N \langle name \rangle: \langle arg spec \rangle`

Defines a quark-test function `\langle name \rangle: \langle arg spec \rangle` which tests if its argument is `\q__\langle namespace \rangle_recursion_tail`, then acts accordingly, as described below for each possible `\langle arg spec \rangle`.

The `\langle namespace \rangle` is determined as the first (nonempty) `_`-delimited word in `\langle name \rangle` and is used internally in the definition of auxiliaries. The function `\__kernel_quark_new_test:N` does *not* define the `\q__\langle namespace \rangle_recursion_tail` and `\q__\langle namespace \rangle_recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the `\langle arg spec \rangle`, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail: (N|n)` and `\quark_if_recursion_tail_do: (N|n)n`.

`n` defines `\langle name \rangle:n` such that it checks if #1 contains only `\q__\langle namespace \rangle_recursion_tail`, and if so consumes all tokens up to `\q__\langle namespace \rangle_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:n`).

`nn` defines `\langle name \rangle:nn` such that it checks if #1 contains only `\q__\langle namespace \rangle_recursion_tail`, and if so consumes all tokens up to `\q__\langle namespace \rangle_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\langle name \rangle:N` such that it checks if #1 is `\q__\langle namespace \rangle_recursion_tail`, and if so consumes all tokens up to `\q__\langle namespace \rangle_recursion_stop` (c.f. `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\langle name \rangle:Nn` such that it checks if #1 is `\q__\langle namespace \rangle_recursion_tail`, and if so consumes all tokens up to `\q__\langle namespace \rangle_recursion_stop`, then executes the code #2 after that (c.f. `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break: (n|N)N`, and in those cases the quark `\q__\langle namespace \rangle_recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\langle name \rangle:nN` such that it checks if #1 contains only `\q__\langle namespace \rangle_recursion_tail`, and if so uses the `\langle type \rangle_map_break: function #2`.

`NN` defines `\langle name \rangle:NN` such that it checks if #1 is `\q__\langle namespace \rangle_recursion_tail`, and if so uses the `\langle type \rangle_map_break: function #2`.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\__kernel_quark_new_conditional:Nn` `\__kernel_quark_new_conditional:Nn \__<namespace>_quark_if_<name>:<arg spec> {<conditions>}`

Defines a collection of quark conditionals that test if their argument is the quark `\q_<namespace>_<name>` and perform suitable actions. The `<conditions>` are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each `<condition>` in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `\__<namespace>_quark_if_<name>:<arg spec>`.

The first argument of `\__kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the `<name>` of the quark `\q_<namespace>_<name>` to be tested. This quark should be manually defined with `\quark_new:N`, as `\__kernel_quark_new_conditional:Nn` does *not* define it.

The function `\__kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the `<arg spec>`, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

`n` defines `\__<namespace>_quark_if_<name>:n(TF)` such that it checks if #1 contains only `\q_<namespace>_<name>`, and executes the proper conditional branch.

`N` defines `\__<namespace>_quark_if_<name>:N(TF)` such that it checks if #1 is `\q_<namespace>_<name>`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\__kernel_sys_everyjob:` `\__kernel_sys_everyjob:`

Inserts the internal token list required at the start of every run (job).

`\c__kernel_randint_max_int` Maximal allowed argument to `\__kernel_randint:n`. Equal to $2^{17} - 1$.

(End of definition for `\c__kernel_randint_max_int`.)

`\__kernel_randint:n` `\__kernel_randint:n {<max>}`

Used in an integer expression this gives a pseudo-random number between 1 and `<max>` included. One must have `<max> ≤ 217 - 1`. The `<max>` must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

`\__kernel_randint:nn` `\__kernel_randint:nn {<min>} {<max>}`

Used in an integer expression this gives a pseudo-random number between `<min>` and `<max>` included. The `<min>` and `<max>` must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$, `<min> - 1 + \__kernel_randint:n{R}` is faster.

`\__kernel_register_show:N` `\__kernel_register_show:N <register>`

`\__kernel_register_show:c` Used to show the contents of a T_EX register at the terminal, formatted such that internal parts of the mechanism are not visible.

`\__kernel_register_log:N` `\__kernel_register_log:N <register>`

`\__kernel_register_log:c` Used to write the contents of a T_EX register to the log file in a form similar to `\__kernel_register_show:N`.

_kernel_str_to_other:n ★ _kernel_str_to_other:n {<token list>}

Converts the <token list> to a <other string>, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

_kernel_str_to_other_fast:n ☆ _kernel_str_to_other_fast:n {<token list>}

Same behavior _kernel_str_to_other:n but only restricted-expandable. It takes a time linear in the character count of the string.

_kernel_tl_to_str:w ★ _kernel_tl_to_str:w <expandable tokens> {<tokens>}

Carries out expansion on the <expandable tokens> before conversion of the <tokens> to a string as describe for \tl_to_str:n. Typically, the <expandable tokens> will alter the nature of the <tokens>, i.e., allow it to be generated in some way. This function requires only a single expansion.

_kernel_tl_set:Nx _kernel_tl_set:Nx <tl var> {<tokens>}

_kernel_tl_gset:Nx

Fully expands <tokens> and assigns the result to <tl var>. <tokens> must be given in braces and there must be no token between <tl var> and <tokens>.

_kernel_codepoint_data:nn ★ _kernel_codepoint_data:nn {<type>} {<codepoint>}

Expands to the appropriate value for the <type> of data requested for a <codepoint>. The current list of <types> and results are

lowercase The *single* codepoint specified by `UnicodeData.txt` for lowercase mapping of the codepoint: will be equal to the input <codepoint> if there is no mapping specified in `UnicodeData.txt`

uppercase The *single* codepoint specified by `UnicodeData.txt` for uppercase mapping of the codepoint: will be equal to the input <codepoint> if there is no mapping specified in `UnicodeData.txt`

_kernel_codepoint_case:nn ★ _kernel_codepoint_case:nn {<mapping>} {<codepoint>}

Expands to a list of three balanced text, of which at least the first will contain a codepoint. This list of up to three codepoints specifies the full case mapping for the input <codepoint>. The <mapping> should be one of

- casefold
- lowercase
- titlecase
- uppercase

45.3 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

```
\_kernel_backend_literal:n \_kernel_backend_literal:n {<content>}
\_kernel_backend_literal:(e|e)
```

Adds the $\langle content \rangle$ literally to the current vertical list as a whatsit. The nature of the $\langle content \rangle$ will depend on the backend in use.

```
\_kernel_backend_literal_postscript:n \_kernel_backend_literal_postscript:n {<PostScript>}
\_kernel_backend_literal_postscript:e
```

Adds the $\langle PostScript \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\_kernel_backend_literal_pdf:n \_kernel_backend_literal_pdf:n {<PDF instructions>}
\_kernel_backend_literal_pdf:e
```

Adds the $\langle PDF instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\_kernel_backend_literal_svg:n \_kernel_backend_literal_svg:n {<SVG instructions>}
\_kernel_backend_literal_svg:e
```

Adds the $\langle SVG instructions \rangle$ literally to the current vertical list as a whatsit. No positioning is applied.

```
\_kernel_backend_postscript:n \_kernel_backend_postscript:n {<PostScript>}
\_kernel_backend_postscript:e
```

Adds the $\langle PostScript \rangle$ to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.

```
\_kernel_backend_align_begin: \_kernel_backend_align_begin:
\_kernel_backend_align_end: <PostScript literals>
\_kernel_backend_align_end:
```

Arranges to align the PostScript and DVI current positions and scales.

```
\_kernel_backend_scope_begin: \_kernel_backend_scope_begin:
\_kernel_backend_scope_end: <content>
\_kernel_backend_scope_end:
```

Creates a scope for instructions at the backend level.

```
\_kernel_backend_matrix:n \_kernel_backend_matrix:n {<matrix>}
\_kernel_backend_matrix:e
```

Applies the $\langle matrix \rangle$ to the current transformation matrix.

```
\g__kernel_backend_header_bool
```

Specifies whether to write headers for the backend.

<code>\l__kernel_color_stack_int</code>	The color stack used in pdfTeX and LuaTeX for the main color.

Chapter 46

l3basics implementation

1385 `(*code)`

46.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁹

<code>\if_true:</code>	Then some conditionals.	
<code>\if_false:</code>		
<code>\or:</code>	1386 <code>\tex_global:D \tex_let:D \if_true:</code>	<code>\tex_iftrue:D</code>
<code>\else:</code>	1387 <code>\tex_global:D \tex_let:D \if_false:</code>	<code>\tex_iffalse:D</code>
<code>\fi:</code>	1388 <code>\tex_global:D \tex_let:D \or:</code>	<code>\tex_or:D</code>
<code>\reverse_if:N</code>	1389 <code>\tex_global:D \tex_let:D \else:</code>	<code>\tex_else:D</code>
<code>\if:w</code>	1390 <code>\tex_global:D \tex_let:D \fi:</code>	<code>\tex_fi:D</code>
<code>\if_charcode:w</code>	1391 <code>\tex_global:D \tex_let:D \reverse_if:N</code>	<code>\tex_unless:D</code>
<code>\if_catcode:w</code>	1392 <code>\tex_global:D \tex_let:D \if:w</code>	<code>\tex_if:D</code>
<code>\if_meaning:w</code>	1393 <code>\tex_global:D \tex_let:D \if_charcode:w</code>	<code>\tex_if:D</code>
	1394 <code>\tex_global:D \tex_let:D \if_catcode:w</code>	<code>\tex_ifcat:D</code>
	1395 <code>\tex_global:D \tex_let:D \if_meaning:w</code>	<code>\tex_ifx:D</code>
	1396 <code>\tex_global:D \tex_let:D \if_bool:N</code>	<code>\tex_ifodd:D</code>

(End of definition for `\if_true:` and others. These functions are documented on page 29.)

<code>\if_mode_math:</code>	T _E X lets us detect some if its modes.	
<code>\if_mode_horizontal:</code>	1397 <code>\tex_global:D \tex_let:D \if_mode_math:</code>	<code>\tex_ifmmode:D</code>
<code>\if_mode_vertical:</code>	1398 <code>\tex_global:D \tex_let:D \if_mode_horizontal:</code>	<code>\tex_ifhmode:D</code>
<code>\if_mode_inner:</code>	1399 <code>\tex_global:D \tex_let:D \if_mode_vertical:</code>	<code>\tex_ifvmode:D</code>
	1400 <code>\tex_global:D \tex_let:D \if_mode_inner:</code>	<code>\tex_ifinner:D</code>

(End of definition for `\if_mode_math:` and others. These functions are documented on page 30.)

<code>\if_cs_exist:N</code>	Building csnames and testing if control sequences exist.	
<code>\if_cs_exist:w</code>		
<code>\cs:w</code>	1401 <code>\tex_global:D \tex_let:D \if_cs_exist:N</code>	<code>\tex_ifdefined:D</code>
<code>\cs_end:</code>	1402 <code>\tex_global:D \tex_let:D \if_cs_exist:w</code>	<code>\tex_ifcurname:D</code>
	1403 <code>\tex_global:D \tex_let:D \cs:w</code>	<code>\tex_csname:D</code>
	1404 <code>\tex_global:D \tex_let:D \cs_end:</code>	<code>\tex_endcurname:D</code>

⁹This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex_...:D` name in the cases where no good alternative exists.

(End of definition for `\if_cs_exist:N` and others. These functions are documented on page 30.)

`\exp_after:wN` The five `\exp_` functions are used in the `l3expan` module where they are described.

```

1405 \tex_global:D \tex_let:D \exp_after:wN      \tex_expandafter:D
1406 \tex_global:D \tex_let:D \exp_not:N         \tex_noexpand:D
1407 \tex_global:D \tex_let:D \exp_not:n         \tex_unexpanded:D
1408 \tex_global:D \tex_let:D \exp:w             \tex_romannumeral:D
1409 \tex_global:D \tex_chardef:D \exp_end:  = 0 ~

```

(End of definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 42.)

`\token_to_meaning:N` Examining a control sequence or token.

```

1410 \tex_global:D \tex_let:D \token_to_meaning:N \tex_meaning:D
1411 \tex_global:D \tex_let:D \cs_meaning:N       \tex_meaning:D

```

(End of definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 206.)

`\tl_to_str:n` Making strings.

```

1412 \tex_global:D \tex_let:D \tl_to_str:n       \tex_detokenize:D
1413 \tex_global:D \tex_let:D \token_to_str:N     \tex_string:D
1414 \tex_global:D \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D

```

(End of definition for `\tl_to_str:n`, `\token_to_str:N`, and `\__kernel_tl_to_str:w`. These functions are documented on page 118.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```

1415 \tex_global:D \tex_let:D \scan_stop:         \tex_relax:D
1416 \tex_global:D \tex_let:D \group_begin:       \tex_begingroup:D
1417 \tex_global:D \tex_let:D \group_end:         \tex_endgroup:D

```

(End of definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 14.)

```

1418 <@@=int>

```

`\if_int_compare:w` For integers.

```

1419 \tex_global:D \tex_let:D \if_int_compare:w    \tex_ifnum:D
1420 \tex_global:D \tex_let:D \__int_to_roman:w    \tex_romannumeral:D

```

(End of definition for `\if_int_compare:w` and `\__int_to_roman:w`. This function is documented on page 185.)

`\group_insert_after:N` Adding material after the end of a group.

```

1421 \tex_global:D \tex_let:D \group_insert_after:N \tex_aftergroup:D

```

(End of definition for `\group_insert_after:N`. This function is documented on page 15.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```

1422 \tex_long:D \tex_gdef:D \exp_args:Nc #1#2
1423 { \exp_after:wN #1 \cs:w #2 \cs_end: }
1424 \tex_long:D \tex_gdef:D \exp_args:cc #1#2
1425 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }

```

(End of definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 38.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:Nnc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1426 \tex_gdef:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1427 \tex_long:D \tex_gdef:D \cs_meaning:c #1
1428 {
1429   \if_cs_exist:w #1 \cs_end:
1430     \exp_after:wN \use_i:nn
1431   \else:
1432     \exp_after:wN \use_ii:nn
1433   \fi:
1434   { \exp_args:Nc \cs_meaning:N {#1} }
1435   { \tl_to_str:n {undefined} }
1436 }
1437 \tex_global:D \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End of definition for `\token_to_meaning:N`. This function is documented on page 206.)

46.2 Defining some constants

`\c_zero_int` We need the constant `\c_zero_int` which is used by some functions in current module. The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can’t be used until the allocation has been set up properly!

```

1438 \tex_global:D \tex_chardef:D \c_zero_int = 0 ~

```

(End of definition for `\c_zero_int`. This variable is documented on page 184.)

`\c_max_register_int` This is here as this particular integer is needed in modules loaded before `l3int`, and is documented in `l3int`. LuaTeX and those which contain parts of the Omega extensions have more registers available than ε -TeX.

```

1439 \tex_ifdefined:D \tex_luatexversion:D
1440   \tex_global:D \tex_chardef:D \c_max_register_int = 65 535 ~
1441 \tex_else:D
1442   \tex_ifdefined:D \tex_omathchardef:D
1443     \tex_global:D \tex_omathchardef:D \c_max_register_int = 65535 ~
1444   \tex_else:D
1445     \tex_global:D \tex_mathchardef:D \c_max_register_int = 32767 ~
1446   \tex_fi:D
1447 \tex_fi:D

```

(End of definition for `\c_max_register_int`. This variable is documented on page 184.)

46.3 Defining functions

We start by providing functions for the typical definition functions. First the global ones.

`\cs_gset_nopar:Npn` All assignment functions in L^AT_EX3 should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren’t.

```

\cs_gset_nopar:Npe
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npe
\cs_gset:Npx
1448 \tex_global:D \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
1449 \tex_global:D \tex_let:D \cs_gset_nopar:Npe \tex_xdef:D

```

```

\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npe
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npe
\cs_gset_protected:Npx

```

```

1450 \tex_global:D \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
1451 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npn
1452 { \tex_long:D \tex_gdef:D }
1453 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npe
1454 { \tex_long:D \tex_xdef:D }
1455 \tex_global:D \tex_let:D \cs_gset:Npx \cs_gset:Npe
1456 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npn
1457 { \tex_protected:D \tex_gdef:D }
1458 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npe
1459 { \tex_protected:D \tex_xdef:D }
1460 \tex_global:D \tex_let:D \cs_gset_protected_nopar:Npx \cs_gset_protected_nopar:Npe
1461 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npn
1462 { \tex_protected:D \tex_long:D \tex_gdef:D }
1463 \tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npe
1464 { \tex_protected:D \tex_long:D \tex_xdef:D }
1465 \tex_global:D \tex_let:D \cs_gset_protected:Npx \cs_gset_protected:Npe

```

(End of definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 18.)

`\cs_set_nopar:Npn` Local versions of the above functions.

```

\cs_set_nopar:Npe
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npe
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npe
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npe
\cs_set_protected:Npx
1466 \tex_global:D \tex_let:D \cs_set_nopar:Npn \tex_def:D
1467 \tex_global:D \tex_let:D \cs_set_nopar:Npe \tex_edef:D
1468 \tex_global:D \tex_let:D \cs_set_nopar:Npx \tex_edef:D
1469 \cs_gset_protected:Npn \cs_set:Npn
1470 { \tex_long:D \tex_def:D }
1471 \cs_gset_protected:Npn \cs_set:Npe
1472 { \tex_long:D \tex_edef:D }
1473 \tex_global:D \tex_let:D \cs_set:Npx \cs_set:Npe
1474 \cs_gset_protected:Npn \cs_set_protected_nopar:Npn
1475 { \tex_protected:D \tex_def:D }
1476 \cs_gset_protected:Npn \cs_set_protected_nopar:Npe
1477 { \tex_protected:D \tex_edef:D }
1478 \tex_global:D \tex_let:D \cs_set_protected_nopar:Npx \cs_set_protected_nopar:Npe
1479 \cs_gset_protected:Npn \cs_set_protected:Npn
1480 { \tex_protected:D \tex_long:D \tex_def:D }
1481 \cs_gset_protected:Npn \cs_set_protected:Npe
1482 { \tex_protected:D \tex_long:D \tex_edef:D }
1483 \tex_global:D \tex_let:D \cs_set_protected:Npx \cs_set_protected:Npe

```

(End of definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 17.)

46.4 Selecting tokens

```

1484 <@@=exp>

```

`\l__exp_tmp_tl` Scratch token list variable for `l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `l3basics` is loaded earlier.

```

1485 \cs_gset_nopar:Npn \l__exp_tmp_tl { }

```

(End of definition for `\l__exp_tmp_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1486 \cs_gset:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End of definition for `\use:c`. This function is documented on page 22.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_tmp_tl` which we’ve set up above.

```
1487 \cs_gset_protected:Npn \use:x #1
1488 {
1489   \cs_set_nopar:Npx \l__exp_tmp_tl {#1}
1490   \l__exp_tmp_tl
1491 }
```

(End of definition for `\use:x`.)

```
1492 <@@=use>
```

`\use:e`

```
1493 \cs_gset:Npn \use:e #1 { \tex_expanded:D {#1} }
```

(End of definition for `\use:e`. This function is documented on page 27.)

```
1494 <@@=exp>
```

`\use:n`

These macros grab their arguments and return them back to the input (with outer braces removed).

`\use:nn`

`\use:nnn`

`\use:nnnn`

```
1495 \cs_gset:Npn \use:n #1 {#1}
1496 \cs_gset:Npn \use:nn #1#2 {#1#2}
1497 \cs_gset:Npn \use:nnn #1#2#3 {#1#2#3}
1498 \cs_gset:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(End of definition for `\use:n` and others. These functions are documented on page 25.)

`\use_i:nn`

`\use_ii:nn`

The equivalent to L^AT_EX 2_ε’s `\@firstoftwo` and `\@secondoftwo`.

```
1499 \cs_gset:Npn \use_i:nn #1#2 {#1}
1500 \cs_gset:Npn \use_ii:nn #1#2 {#2}
```

(End of definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 26.)

`\use_i:nnn`

`\use_ii:nnn`

`\use_iii:nnn`

`\use_i:nnnn`

`\use_ii:nnnn`

`\use_iii:nnnn`

`\use_iv:nnnn`

`\use_i:nnnnn`

`\use_ii:nnnnn`

`\use_iii:nnnnn`

`\use_iv:nnnnn`

`\use_v:nnnnn`

`\use_i:nnnnnn`

`\use_ii:nnnnnn`

`\use_iii:nnnnnn`

`\use_iv:nnnnnn`

`\use_v:nnnnnn`

`\use_vi:nnnnnn`

`\use_i:nnnnnnn`

`\use_ii:nnnnnnn`

`\use_iii:nnnnnnn`

`\use_iv:nnnnnnn`

`\use_v:nnnnnnn`

`\use_vi:nnnnnnn`

`\use_vii:nnnnnnn`

`\use_i:nnnnnnnn`

`\use_ii:nnnnnnnn`

`\use_iii:nnnnnnnn`

`\use_iv:nnnnnnnn`

`\use_v:nnnnnnnn`

We also need something for picking up arguments from a longer list.

```
1501 \cs_gset:Npn \use_i:nnn #1#2#3 {#1}
1502 \cs_gset:Npn \use_ii:nnn #1#2#3 {#2}
1503 \cs_gset:Npn \use_iii:nnn #1#2#3 {#3}
1504 \cs_gset:Npn \use_i:nnnn #1#2#3#4 {#1}
1505 \cs_gset:Npn \use_ii:nnnn #1#2#3#4 {#2}
1506 \cs_gset:Npn \use_iii:nnnn #1#2#3#4 {#3}
1507 \cs_gset:Npn \use_iv:nnnn #1#2#3#4 {#4}
1508 \cs_gset:Npn \use_i:nnnnn #1#2#3#4#5 {#1}
1509 \cs_gset:Npn \use_ii:nnnnn #1#2#3#4#5 {#2}
1510 \cs_gset:Npn \use_iii:nnnnn #1#2#3#4#5 {#3}
1511 \cs_gset:Npn \use_iv:nnnnn #1#2#3#4#5 {#4}
1512 \cs_gset:Npn \use_v:nnnnn #1#2#3#4#5 {#5}
1513 \cs_gset:Npn \use_i:nnnnnn #1#2#3#4#5#6 {#1}
1514 \cs_gset:Npn \use_ii:nnnnnn #1#2#3#4#5#6 {#2}
1515 \cs_gset:Npn \use_iii:nnnnnn #1#2#3#4#5#6 {#3}
1516 \cs_gset:Npn \use_iv:nnnnnn #1#2#3#4#5#6 {#4}
1517 \cs_gset:Npn \use_v:nnnnnn #1#2#3#4#5#6 {#5}
1518 \cs_gset:Npn \use_vi:nnnnnn #1#2#3#4#5#6 {#6}
1519 \cs_gset:Npn \use_i:nnnnnnn #1#2#3#4#5#6#7 {#1}
1520 \cs_gset:Npn \use_ii:nnnnnnn #1#2#3#4#5#6#7 {#2}
1521 \cs_gset:Npn \use_iii:nnnnnnn #1#2#3#4#5#6#7 {#3}
```

```

1522 \cs_gset:Npn \use_iv:nnnnnnnn #1#2#3#4#5#6#7 {#4}
1523 \cs_gset:Npn \use_v:nnnnnnnn #1#2#3#4#5#6#7 {#5}
1524 \cs_gset:Npn \use_vi:nnnnnnnn #1#2#3#4#5#6#7 {#6}
1525 \cs_gset:Npn \use_vii:nnnnnnnn #1#2#3#4#5#6#7 {#7}
1526 \cs_gset:Npn \use_i:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#1}
1527 \cs_gset:Npn \use_ii:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#2}
1528 \cs_gset:Npn \use_iii:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#3}
1529 \cs_gset:Npn \use_iv:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#4}
1530 \cs_gset:Npn \use_v:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#5}
1531 \cs_gset:Npn \use_vi:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#6}
1532 \cs_gset:Npn \use_vii:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#7}
1533 \cs_gset:Npn \use_viii:nnnnnnnnnn #1#2#3#4#5#6#7#8 {#8}
1534 \cs_gset:Npn \use_i:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#1}
1535 \cs_gset:Npn \use_ii:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#2}
1536 \cs_gset:Npn \use_iii:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#3}
1537 \cs_gset:Npn \use_iv:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#4}
1538 \cs_gset:Npn \use_v:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#5}
1539 \cs_gset:Npn \use_vi:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#6}
1540 \cs_gset:Npn \use_vii:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#7}
1541 \cs_gset:Npn \use_viii:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#8}
1542 \cs_gset:Npn \use_ix:nnnnnnnnnnnn #1#2#3#4#5#6#7#8#9 {#9}

```

(End of definition for \use_i:nnn and others. These functions are documented on page 26.)

\use_i_ii:nnn

```

1543 \cs_gset:Npn \use_i_ii:nnn #1#2#3 {#1#2}

```

(End of definition for \use_i_ii:nnn. This function is documented on page 27.)

\use_ii_i:nn

```

1544 \cs_gset:Npn \use_ii_i:nn #1#2 { #2 #1 }

```

(End of definition for \use_ii_i:nn. This function is documented on page 27.)

\use_none_delimit_by_q_nil:w Functions that gobble everything until they see either \q_nil, \q_stop, or \q_recursion_stop, respectively.

```

\use_none_delimit_by_q_stop:w
\use_none_delimit_by_q_recursion_stop:w
1545 \cs_gset:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1546 \cs_gset:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1547 \cs_gset:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }

```

(End of definition for \use_none_delimit_by_q_nil:w, \use_none_delimit_by_q_stop:w, and \use_none_delimit_by_q_recursion_stop:w. These functions are documented on page 27.)

\use_i_delimit_by_q_nil:nw Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

\use_i_delimit_by_q_stop:nw

```

\use_i_delimit_by_q_recursion_stop:nw
1548 \cs_gset:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1549 \cs_gset:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1550 \cs_gset:Npn \use_i_delimit_by_q_recursion_stop:nw
1551 #1#2 \q_recursion_stop {#1}

```

(End of definition for \use_i_delimit_by_q_nil:nw, \use_i_delimit_by_q_stop:nw, and \use_i_delimit_by_q_recursion_stop:nw. These functions are documented on page 28.)

46.5 Gobbling tokens from input

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of n's following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:n`, `\use_none:nn`, `\use_none:nnn`, `\use_none:nnnn`, `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

(End of definition for \use none:n and others. These functions are documented on page 27.)

46.6 Debugging and patching later definitions

`\_kernel_if_debug:TF` A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```
1562 \cs_gset_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
```

(End of definition for `\_kernel_if_debug:TF`.)

```
\debug_on:n Stubs.
```

```

1563 \cs_gset_protected:Npn \debug_on:n #1
1564 {
1565     \sys_load_debug:
1566     \cs_if_exist:NT \__debug_all_on:
1567     { \debug_on:n {#1} }
1568 }
1569 \cs_gset_protected:Npn \debug_off:n #1
1570 {
1571     \sys_load_debug:
1572     \cs_if_exist:NT \__debug_all_on:
1573     { \debug_off:n {#1} }
1574 }

```

(End of definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 31.)

```
\debug_suspend:
```

```
\debug_resume: 1575 \cs_gset_protected:Npn \debug_suspend: { }
1576 \cs_gset_protected:Npn \debug_resume: { }
```

(End of definition for `\debug_suspend:` and `\debug_resume:`. These functions are documented on page 31.)

```

\debug_assert:nN
\debug_assert:nn
\debug_assert:nNn
\debug_assert:nnn
1577 \cs_gset:Npn \debug_assert:nN #1#2 { }
1578 \cs_gset:Npn \debug_assert:nn #1#2 { }
1579 \cs_gset:Npn \debug_assert:nNn #1#2#3 { }
1580 \cs_gset:Npn \debug_assert:nnn #1#2#3 { }

```

(End of definition for `\debug_assert:nN` and others. These functions are documented on page 32.)

```

\__kernel_deprecation_code:nn
\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl
1581 \cs_gset_nopar:Npn \g__debug_deprecation_on_tl { }
1582 \cs_gset_nopar:Npn \g__debug_deprecation_off_tl { }
1583 \cs_gset_protected:Npn \__kernel_deprecation_code:nn #1#2
1584 {
1585   \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
1586   \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
1587 }

```

(End of definition for `\__kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

46.7 Conditional processing and definitions

```

1588 \@@=prg\

```

Underneath any predicate function (`\_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves `\TeX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2
\prg_return_true:
\else:
\if_meaning:w #1#3
\prg_return_true:
\else:
\prg_return_false:
\fi:
\fi:

```

Usually, a `\TeX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `\TeX` programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```

1589 \cs_gset:Npn \prg_return_true:
1590 { \exp_after:wN \use_i:nn \exp:w }
1591 \cs_gset:Npn \prg_return_false:
1592 { \exp_after:wN \use_ii:nn \exp:w}

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(End of definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 67.)

`\prg_use_none_delimit_by_q_recursion_stop:w`

Private version of `\use_none_delimit_by_q_recursion_stop:w`.

```
1593 \cs_gset:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
1594   #1 \q__prg_recursion_stop { }
```

(End of definition for `\__prg_use_none_delimit_by_q_recursion_stop:w`.)

`\prg_set_conditional:Npnn`
`\prg_gset_conditional:Npnn`
`\prg_new_conditional:Npnn`
`\prg_set_protected_conditional:Npnn`
`\prg_gset_protected_conditional:Npnn`
`\prg_new_protected_conditional:Npnn`
`\__prg_generate_conditional_parm:NNNpnn`

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{\text{TF}, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. Note that `e` stands for expandable and `p` for protected.

```
1595 \cs_gset_protected:Npn \prg_set_conditional:Npnn
1596   { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1597 \cs_gset_protected:Npn \prg_gset_conditional:Npnn
1598   { \__prg_generate_conditional_parm:NNNpnn \cs_gset:Npn e }
1599 \cs_gset_protected:Npn \prg_new_conditional:Npnn
1600   { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1601 \cs_gset_protected:Npn \prg_set_protected_conditional:Npnn
1602   { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1603 \cs_gset_protected:Npn \prg_gset_protected_conditional:Npnn
1604   { \__prg_generate_conditional_parm:NNNpnn \cs_gset_protected:Npn p }
1605 \cs_gset_protected:Npn \prg_new_protected_conditional:Npnn
1606   { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1607 \cs_gset_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1608   {
1609     \use:e
1610     {
1611       \__prg_generate_conditional:nnNNNnnn
1612       \cs_split_function:N #3
1613     }
1614     #1 #2 {#4}
1615   }
```

(End of definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 66.)

`\prg_set_conditional:Nnn`
`\prg_gset_conditional:Nnn`
`\prg_new_conditional:Nnn`
`\prg_set_protected_conditional:Nnn`
`\prg_gset_protected_conditional:Nnn`
`\prg_new_protected_conditional:Nnn`
`\__prg_generate_conditional_count:NNNnn`
`\__prg_generate_conditional_count:nnNNNnn`

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed $\{\langle name \rangle\}$ $\{\langle signature \rangle\}$ $\langle boolean \rangle$ $\{\langle set \text{ or } new \rangle\}$ $\{\langle maybe \text{ protected} \rangle\}$ $\{\langle parameters \rangle\}$ $\{\text{TF}, \dots\}$ $\{\langle code \rangle\}$ to the auxiliary function responsible for defining all conditionals. If the $\langle signature \rangle$ has more than 9 letters, the definition is aborted since $\text{\TeX}$ macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.


```

1616 \cs_gset_protected:Npn \prg_set_conditional:Nnn
1617 { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1618 \cs_gset_protected:Npn \prg_gset_conditional:Nnn
1619 { \__prg_generate_conditional_count:NNNnn \cs_gset:Npn e }
1620 \cs_gset_protected:Npn \prg_new_conditional:Nnn
1621 { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
1622 \cs_gset_protected:Npn \prg_set_protected_conditional:Nnn
1623 { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1624 \cs_gset_protected:Npn \prg_gset_protected_conditional:Nnn
1625 { \__prg_generate_conditional_count:NNNnn \cs_gset_protected:Npn p }
1626 \cs_gset_protected:Npn \prg_new_protected_conditional:Nnn
1627 { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1628 \cs_gset_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
1629 {
1630   \use:e
1631   {
1632     \__prg_generate_conditional_count:nnNNNnn
1633     \cs_split_function:N #3
1634   }
1635   #1 #2
1636 }
1637 \cs_gset_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1638 {
1639   \__kernel_cs_parm_from_arg_count:nnF
1640   { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
1641   { \tl_count:n {#2} }
1642   {
1643     \msg_error:nnee { kernel } { bad-number-of-arguments }
1644     { \token_to_str:c { #1 : #2 } }
1645     { \tl_count:n {#2} }
1646     \use_none:nn
1647   }
1648 }

```

(End of definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 66.)

```

\__prg_generate_conditional:nnNNNnnn
\__prg_generate_conditional:NNnnnnNw
\__prg_generate_conditional_test:w
\__prg_generate_conditional_fast:nw

```

The workhorse here is going through a list of desired forms, i.e., p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

A large number of our low-level conditionals look like `<code> \prg_return_true:\else: \prg_return_false: \fi:` so we optimize this special case by calling `\__prg_generate_conditional_fast:nw {<code>}`. This passes `\use_i:nn` instead of `\use_i_ii:nnn` to functions such as `\__prg_generate_p_form:wNNnnnnN`.

```

1649 \cs_gset_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1650 {
1651   \if_meaning:w \c_false_bool #3
1652   \msg_error:nne { kernel } { missing-colon }

```

```

1653     { \token_to_str:c {#1} }
1654     \exp_after:wN \use_none:nn
1655   \fi:
1656   \use:e
1657   {
1658     \exp_not:N \__prg_generate_conditional:NNnnnnNw
1659     \exp_not:n { #4 #5 {#1} {#2} {#6} }
1660     \__prg_generate_conditional_test:w
1661     #8 \s__prg_mark
1662     \__prg_generate_conditional_fast:nw
1663     \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
1664     \use_none:n
1665     \exp_not:n { {#8} \use_i_ii:nnn }
1666     \tl_to_str:n {#7}
1667     \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1668   }
1669 }
1670 \cs_gset:Npn \__prg_generate_conditional_test:w
1671   #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
1672   { #2 {#1} }
1673 \cs_gset:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1674   { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```

1675 \cs_gset_protected:Npn \__prg_generate_conditional:NNnnnnNw #1#2#3#4#5#6#7#8 ,
1676 {
1677   \if_meaning:w \q__prg_recursion_tail #8
1678   \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1679   \fi:
1680   \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1681   \tl_if_empty:nF {#8}
1682   {
1683     \msg_error:nnee
1684     { kernel } { conditional-form-unknown }
1685     {#8} { \token_to_str:c { #3 : #4 } }
1686   }
1687   \use_none:nnnnnnnn
1688   \s__prg_stop
1689   #1 #2 {#3} {#4} {#5} {#6} #7
1690   \__prg_generate_conditional:NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1691 }

```

(End of definition for `\__prg_generate_conditional:nnNNnnnn` and others.)

```

\__prg_generate_p_form:wNNnnnnN
\__prg_generate_TF_form:wNNnnnnN
\__prg_generate_T_form:wNNnnnnN
\__prg_generate_F_form:wNNnnnnN
  \__prg_p_true:w
  \__prg_T_true:w
  \__prg_F_true:w
  \__prg_TF_true:w

```

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: `\cs_set:Npn` or similar, 3: `p` (for protected conditionals) or `e`, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by `\__prg_generate_conditional_fast:nw`), 8: `\use_i_ii:nnn` or `\use_i:nn` (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present after `\exp_end::` notice the construction of the different variants relies on this, and that

the TF and F variants will be slightly faster than the T version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, #7 has an extra `\if_...`. To optimize a bit further we don’t use `\exp_after:wN \use_ii:nnn` and similar but instead use `\__prg_TF_true:w` and similar to swap out the macro after `\fi:`. It would be a tiny bit faster if we directly grabbed the T and F arguments there, but if those are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```

1692 \cs_gset_protected:Npn \__prg_generate_p_form:wNNnnnnN
1693   #1 \s__prg_stop #2#3#4#5#6#7#8
1694   {
1695     \if_meaning:w e #3
1696     \exp_after:wN \use_i:nn
1697     \else:
1698     \exp_after:wN \use_ii:nn
1699     \fi:
1700     {
1701       #8
1702       { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1703       { { #7 \exp_end: \c_true_bool \c_false_bool } }
1704       { #7 \__prg_p_true:w \fi: \c_false_bool }
1705     }
1706     {
1707       \msg_error:nne { kernel } { protected-predicate }
1708       { \token_to_str:c { #4 _p: #5 } }
1709     }
1710   }
1711 \cs_gset_protected:Npn \__prg_generate_T_form:wNNnnnnN
1712   #1 \s__prg_stop #2#3#4#5#6#7#8
1713   {
1714     #8
1715     { \exp_args:Nc #2 { #4 : #5 T } #6 }
1716     { { #7 \exp_end: \use:n \use_none:n } }
1717     { #7 \__prg_T_true:w \fi: \use_none:n }
1718   }
1719 \cs_gset_protected:Npn \__prg_generate_F_form:wNNnnnnN
1720   #1 \s__prg_stop #2#3#4#5#6#7#8
1721   {
1722     #8
1723     { \exp_args:Nc #2 { #4 : #5 F } #6 }
1724     { { #7 \exp_end: { } } }
1725     { #7 \__prg_F_true:w \fi: \use:n }
1726   }
1727 \cs_gset_protected:Npn \__prg_generate_TF_form:wNNnnnnN
1728   #1 \s__prg_stop #2#3#4#5#6#7#8
1729   {
1730     #8
1731     { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1732     { { #7 \exp_end: { } } }
1733     { #7 \__prg_TF_true:w \fi: \use_ii:nn }
1734   }
1735 \cs_gset:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
1736 \cs_gset:Npn \__prg_T_true:w \fi: \use_none:n { \fi: \use:n }
1737 \cs_gset:Npn \__prg_F_true:w \fi: \use:n { \fi: \use_none:n }

```

```
1738 \cs_gset:Npn \__prg_TF_true:w \fi: \use_ii:nn { \fi: \use_i:nn }
```

(End of definition for __prg_generate_p_form:wNNnnnnN and others.)

\prg_set_eq_conditional:NNn The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\}$
\prg_gset_eq_conditional:NNn $\langle boolean_1 \rangle \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\} \langle boolean_2 \rangle \langle copying\ function \rangle \langle conditions \rangle$,
\prg_new_eq_conditional:NNn \q__prg_recursion_tail , \q__prg_recursion_stop to a first auxiliary.
__prg_set_eq_conditional:NNNn

```
1739 \cs_gset_protected:Npn \prg_set_eq_conditional:NNn
1740 { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1741 \cs_gset_protected:Npn \prg_gset_eq_conditional:NNn
1742 { \__prg_set_eq_conditional:NNNn \cs_gset_eq:cc }
1743 \cs_gset_protected:Npn \prg_new_eq_conditional:NNn
1744 { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1745 \cs_gset_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1746 {
1747   \use:e
1748   {
1749     \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1750     \cs_split_function:N #2
1751     \cs_split_function:N #3
1752     \exp_not:N #1
1753     \tl_to_str:n {#4}
1754     \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1755   }
1756 }
```

(End of definition for \prg_set_eq_conditional:NNn and others. These functions are documented on page 67.)

__prg_set_eq_conditional:nnNnnNNw
__prg_set_eq_conditional_loop:nnnnNw
__prg_set_eq_conditional_p_form:nnn
__prg_set_eq_conditional_TF_form:nnn
__prg_set_eq_conditional_T_form:nnn
__prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments $\{\langle name_1 \rangle\} \{\langle signature_1 \rangle\} \{\langle name_2 \rangle\} \{\langle signature_2 \rangle\}$ $\langle copying\ function \rangle$ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
1757 \cs_gset_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1758 {
1759   \if_meaning:w \c_false_bool #3
1760   \msg_error:nne { kernel } { missing-colon }
1761   { \token_to_str:c {#1} }
1762   \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1763   \fi:
1764   \if_meaning:w \c_false_bool #6
1765   \msg_error:nne { kernel } { missing-colon }
1766   { \token_to_str:c {#4} }
1767   \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1768   \fi:
1769   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1770 }
1771 \cs_gset_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1772 {
1773   \if_meaning:w \q__prg_recursion_tail #6
1774   \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
```

```

1775 \fi:
1776 \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1777 \tl_if_empty:nF {#6}
1778 {
1779 \msg_error:nnee
1780 { kernel } { conditional-form-unknown }
1781 {#6} { \token_to_str:c { #1 : #2 } }
1782 }
1783 \use_none:nnnnnn
1784 \s__prg_stop
1785 #5 {#1} {#2} {#3} {#4}
1786 \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1787 }
1788 \cs_gset:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1789 { #2 { #3 _p : #4 } { #5 _p : #6 } }
1790 \cs_gset:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1791 { #2 { #3 : #4 TF } { #5 : #6 TF } }
1792 \cs_gset:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1793 { #2 { #3 : #4 T } { #5 : #6 T } }
1794 \cs_gset:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
1795 { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End of definition for __prg_set_eq_conditional:nnNnnNw and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using \if:w but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

\c_true_bool
\c_false_bool

Here are the canonical boolean values.

```

1796 \tex_global:D \tex_chardef:D \c_true_bool = 1 ~
1797 \tex_global:D \tex_chardef:D \c_false_bool = 0 ~

```

(End of definition for \c_true_bool and \c_false_bool. These variables are documented on page 69.)

46.8 Dissecting a control sequence

```

1798 <@@=cs>

```

__cs_count_signature:N ★ __cs_count_signature:N <function>

Splits the <function> into the <name> (i.e., the part before the colon) and the <signature> (i.e., after the colon). The <number> of tokens in the <signature> is then left in the input stream. If there was no <signature> then the result is the marker value -1.

__cs_tmp:w Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\cs_to_str:N` This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape character, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N \_` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `\_cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a space from `\token_to_str:N \_`, and the auxiliary `\_cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `\_cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1799 \cs_gset:Npn \cs_to_str:N
1800 {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
1801 \tex_romannumeral:D
1802 \if:w \token_to_str:N \\_cs_to_str:w \fi:
1803 \exp_after:wN \_cs_to_str:N \token_to_str:N
1804 }
1805 \cs_gset:Npn \_cs_to_str:N #1 { \c_zero_int }
1806 \cs_gset:Npn \_cs_to_str:w #1 \_cs_to_str:N
1807 { - \int_value:w \fi: \exp_after:wN \c_zero_int }
```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End of definition for `\cs_to_str:N`, `\_cs_to_str:N`, and `\_cs_to_str:w`. This function is documented on page 23.)

`\cs_split_function:N` This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as #1 the function name, delimited by the first colon, then the signature #2, delimited by `\s__cs_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\s__cs_stop`. Otherwise, the #1 contains the function name and `\s__cs_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1808 \cs_gset_protected:Npn \__cs_tmp:w #1
1809 {
1810   \cs_gset:Npn \cs_split_function:N ##1
1811   {
1812     \exp_after:wN \exp_after:wN \exp_after:wN
1813     \__cs_split_function_auxi:w
1814     \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1815     #1 \s__cs_mark \c_false_bool \s__cs_stop
1816   }
1817   \cs_gset:Npn \__cs_split_function_auxi:w
1818   ##1 #1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1819   { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1820   \cs_gset:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1821   { {##1} }
1822 }
1823 \exp_after:wN \__cs_tmp:w \token_to_str:N :

```

(End of definition for `\cs_split_function:N`, `\__cs_split_function_auxi:w`, and `\__cs_split_function_auxii:w`. This function is documented on page 24.)

46.9 Exist or free

A control sequence is said to *exist* (to be used) if it has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N` Two versions for checking existence. For the N form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.
`\cs_if_exist_p:c`
`\cs_if_exist:NTF`
`\cs_if_exist:cTF`
`\__cs_if_exist_c_aux:`
`\__cs_if_exist_c_aux:w`

```

1824 \prg_gset_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1825 {
1826   \if_meaning:w #1 \scan_stop:
1827   \use_i:nnnn
1828   \else:
1829   \fi:
1830   \if_cs_exist:N #1
1831   \prg_return_true:
1832   \else:

```

```

1833     \prg_return_false:
1834   \fi:
1835 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1836 \cs_if_exist:NTF \tex_lastnamedcs:D
1837 {
1838   \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1839   {
1840     \if_cs_exist:w #1 \cs_end:
1841       \__cs_if_exist_c_aux:
1842       \prg_return_true:
1843     \else:
1844       \prg_return_false:
1845     \fi:
1846   }
1847   \cs_gset:Npn \__cs_if_exist_c_aux:
1848     { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: \else: }
1849 }
1850 {
1851   \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1852   {
1853     \if_cs_exist:w #1 \cs_end:
1854       \__cs_if_exist_c_aux:w
1855     \fi:
1856     \use_none:n {#1}
1857     \if_false:
1858       \prg_return_true:
1859     \else:
1860       \prg_return_false:
1861     \fi:
1862   }
1863   \cs_gset:Npn \__cs_if_exist_c_aux:w \fi: \use_none:n #1 \if_false:
1864     { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: \else: }
1865 }

```

(End of definition for `\cs_if_exist:NTF`, `\__cs_if_exist_c_aux:`, and `\__cs_if_exist_c_aux:w`. This function is documented on page 29.)

`\cs_if_free_p:N` The logical reversal of the above.

`\cs_if_free_p:c`

`\cs_if_free:NTF`

`\cs_if_free:cTF`

```

1866 \prg_gset_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1867 {
1868   \if_cs_exist:N #1
1869   \else:
1870     \use_none:nnnn
1871   \fi:
1872   \if_meaning:w #1 \scan_stop:
1873     \prg_return_true:
1874   \else:
1875     \prg_return_false:

```



```

1876     \fi:
1877   }
1878 \cs_if_exist:NTF \tex_lastnamedcs:D
1879 {
1880   \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1881   {
1882     \if_cs_exist:w #1 \cs_end:
1883     \__cs_if_free_c_aux:w
1884     \fi:
1885     \if_true:
1886       \prg_return_true:
1887     \else:
1888       \prg_return_false:
1889     \fi:
1890   }
1891   \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \if_true:
1892   { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: }
1893 }
1894 {
1895   \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1896   {
1897     \if_cs_exist:w #1 \cs_end:
1898     \__cs_if_free_c_aux:w
1899     \fi:
1900     \use_none:n {#1}
1901     \if_true:
1902       \prg_return_true:
1903     \else:
1904       \prg_return_false:
1905     \fi:
1906   }
1907   \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \use_none:n #1 \if_true:
1908   { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: }
1909 }

```

(End of definition for \cs_if_free:NTF. This function is documented on page 29.)

\cs_if_exist_use:N The \cs_if_exist_use:... functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the c variants, we are careful not to put the control sequence in the hash table if it does not exist. If available we use the \lastnamedcs primitive.

\cs_if_exist_use:c

\cs_if_exist_use:NTF

\cs_if_exist_use:cTF

```

\__cs_if_exist_use_aux:w 1910 \cs_gset:Npn \cs_if_exist_use:NTF #1#2
\__cs_if_exist_use_aux:Nnn 1911 { \cs_if_exist:NTF #1 { #1 #2 } }
1912 \cs_gset:Npn \cs_if_exist_use:NF #1
1913 { \cs_if_exist:NTF #1 #1 }
1914 \cs_gset:Npn \cs_if_exist_use:NT #1 #2
1915 { \cs_if_exist:NT #1 { #1 #2 } }
1916 \cs_gset:Npn \cs_if_exist_use:N #1
1917 { \cs_if_exist:NT #1 #1 }
1918 \cs_if_exist:NTF \tex_lastnamedcs:D
1919 {
1920   \cs_gset:Npn \cs_if_exist_use:cTF #1
1921   {
1922     \if_cs_exist:w #1 \cs_end:

```

```

1923     \cs_if_exist_use_aux:w
1924     \fi:
1925     \use_ii:nn
1926   }
1927   \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_ii:nn
1928   { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \tex_lastnamedcs:D }
1929 }
1930 {
1931   \cs_gset:Npn \cs_if_exist_use:cTF #1
1932   {
1933     \if_cs_exist:w #1 \cs_end:
1934     \__cs_if_exist_use_aux:w
1935     \fi:
1936     \use_iii:nnn {#1}
1937   }
1938   \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_iii:nnn #1
1939   { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \cs:w #1 \cs_end: }
1940 }
1941 \cs_gset:Npn \__cs_if_exist_use_aux:Nnn #1#2
1942 {
1943   \if_meaning:w #1 \scan_stop:
1944   \exp_after:wN \use_iii:nnn
1945   \fi:
1946   \use_i:nn { #1 #2 }
1947 }
1948 \cs_gset:Npn \cs_if_exist_use:cF #1
1949 { \cs_if_exist_use:cTF {#1} {} }
1950 \cs_gset:Npn \cs_if_exist_use:cT #1#2
1951 { \cs_if_exist_use:cTF {#1} {#2} {} }
1952 \cs_gset:Npn \cs_if_exist_use:c #1
1953 { \cs_if_exist_use:cTF {#1} {} {} }

```

(End of definition for `\cs_if_exist_use:NTF`, `\__cs_if_exist_use_aux:w`, and `\__cs_if_exist_use_aux:Nnn`. This function is documented on page 23.)

46.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`\msg_error:nnee` If an internal error occurs before L^AT_EX3 has loaded l3msg then the code should issue a
`\msg_error:nne` usable if terse error message and halt. This can only happen if a coding error is made by
`\msg_error:nn` the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn
`^^J` into a proper line break in plain T_EX.

```

1954 \cs_gset_protected:Npn \msg_error:nnee #1#2#3#4
1955 {
1956   \tex_newlinechar:D = '\^^J \scan_stop:
1957   \tex_errmessage:D

```

```

1958     {
1959         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1960         Argh,~internal~LaTeX3~error! ^^J ^^J
1961         Module ~ #1 , ~ message~name~"#2": ^^J
1962         Arguments~'#3'~and~'#4' ^^J ^^J
1963         This~is~one~for~The~LaTeX3~Project:~bailing~out
1964     }
1965     \tex_end:D
1966 }
1967 \cs_gset_protected:Npn \msg_error:nne #1#2#3
1968 { \msg_error:nnee {#1} {#2} {#3} { } }
1969 \cs_gset_protected:Npn \msg_error:nn #1#2
1970 { \msg_error:nnee {#1} {#2} { } { } }

```

(End of definition for \msg_error:nnnn. This function is documented on page 87.)

\msg_line_context: Another one from l3msg which will be altered later.

```

1971 \cs_gset:Npn \msg_line_context:
1972 { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End of definition for \msg_line_context:. This function is documented on page 85.)

__kernel_chk_if_free_cs:N
__kernel_chk_if_free_cs:c

This command is called by \cs_new_nopar:Npn and \cs_new_eq:NN etc. to make sure that the argument sequence is not already in use. If it is, an error is signaled. It checks if $\langle csname \rangle$ is undefined or \scan_stop:. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an \if... type function!

```

1973 \cs_gset_protected:Npn \__kernel_chk_if_free_cs:N #1
1974 {
1975     \cs_if_free:NF #1
1976     {
1977         \msg_error:nnee { kernel } { command-already-defined }
1978         { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1979     }
1980 }
1981 \cs_gset_protected:Npn \__kernel_chk_if_free_cs:c
1982 { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End of definition for __kernel_chk_if_free_cs:N.)

46.11 Defining new functions

```

1983 <@@=cs>

```

\cs_new_nopar:Npn Function which check that the control sequence is free before defining it.

\cs_new_nopar:Npe

\cs_new_nopar:Npx

\cs_new:Npn

\cs_new:Npe

\cs_new:Npx

\cs_new_protected_nopar:Npn

\cs_new_protected_nopar:Npe

\cs_new_protected_nopar:Npx

\cs_new_protected:Npn

\cs_new_protected:Npe

\cs_new_protected:Npx

__cs_tmp:w

```

1984 \cs_set:Npn \__cs_tmp:w #1#2
1985 {
1986     \cs_gset_protected:Npn #1 ##1
1987     {
1988         \__kernel_chk_if_free_cs:N ##1
1989         #2 ##1
1990     }
1991 }
1992 \__cs_tmp:w \cs_new_nopar:Npn          \cs_gset_nopar:Npn

```

```

1993 \__cs_tmp:w \cs_new_nopar:Npe \cs_gset_nopar:Npe
1994 \__cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
1995 \__cs_tmp:w \cs_new:Npn \cs_gset:Npn
1996 \__cs_tmp:w \cs_new:Npe \cs_gset:Npe
1997 \__cs_tmp:w \cs_new:Npx \cs_gset:Npx
1998 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1999 \__cs_tmp:w \cs_new_protected_nopar:Npe \cs_gset_protected_nopar:Npe
2000 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
2001 \__cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
2002 \__cs_tmp:w \cs_new_protected:Npe \cs_gset_protected:Npe
2003 \__cs_tmp:w \cs_new_protected:Npx \cs_gset_protected:Npx

```

(End of definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 16.)

`\cs_set_nopar:cpn` Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for csname argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn⟨string⟩⟨rep-text⟩` turns `⟨string⟩` into a csname and then assigns `⟨rep-text⟩` to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

2004 \cs_set:Npn \__cs_tmp:w #1#2
2005 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
2006 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
2007 \__cs_tmp:w \cs_set_nopar:cpe \cs_set_nopar:Npe
2008 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
2009 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
2010 \__cs_tmp:w \cs_gset_nopar:cpe \cs_gset_nopar:Npe
2011 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
2012 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
2013 \__cs_tmp:w \cs_new_nopar:cpe \cs_new_nopar:Npe
2014 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End of definition for `\cs_set_nopar:Npn`. This function is documented on page 17.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments.

`\cs_set:cpe` We may also do this globally.

`\cs_set:cpx`

```

2015 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
2016 \__cs_tmp:w \cs_set:cpe \cs_set:Npe
2017 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
2018 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
2019 \__cs_tmp:w \cs_gset:cpe \cs_gset:Npe
2020 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
2021 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
2022 \__cs_tmp:w \cs_new:cpe \cs_new:Npe
2023 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End of definition for `\cs_set:Npn`. This function is documented on page 17.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

`\cs_set_protected_nopar:cpe`

`\cs_set_protected_nopar:cpx`

```

2024 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
2025 \__cs_tmp:w \cs_set_protected_nopar:cpe \cs_set_protected_nopar:Npe
2026 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx

```

`\cs_gset_protected_nopar:cpn`

`\cs_gset_protected_nopar:cpe`

`\cs_gset_protected_nopar:cpx`

`\cs_new_protected_nopar:cpn`

`\cs_new_protected_nopar:cpe`

`\cs_new_protected_nopar:cpx`

```

2027 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
2028 \__cs_tmp:w \cs_gset_protected_nopar:cpe \cs_gset_protected_nopar:Npe
2029 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
2030 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
2031 \__cs_tmp:w \cs_new_protected_nopar:cpe \cs_new_protected_nopar:Npe
2032 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End of definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 18.)

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```

\cs_set_protected:cpe
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpe
\cs_gset_protected:cpx
\cs_new_protected:cpn
\cs_new_protected:cpe
\cs_new_protected:cpx
2033 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
2034 \__cs_tmp:w \cs_set_protected:cpe \cs_set_protected:Npe
2035 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
2036 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
2037 \__cs_tmp:w \cs_gset_protected:cpe \cs_gset_protected:Npe
2038 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
2039 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
2040 \__cs_tmp:w \cs_new_protected:cpe \cs_new_protected:Npe
2041 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End of definition for `\cs_set_protected:Npn`. This function is documented on page 17.)

46.12 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.

`\cs_set_eq:cN` The = sign allows us to define funny char tokens like = itself or `\_` with this function.

`\cs_set_eq:Nc` For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_set_eq:cc` `\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While

`\cs_gset_eq:NN` `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it

`\cs_gset_eq:cN` long in order to throw an “already defined” error rather than “runaway argument”.

```

\cs_gset_eq:Nc
\cs_gset_eq:cc
\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc
2042 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
2043 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
2044 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
2045 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
2046 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
2047 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
2048 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
2049 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
2050 \cs_new_protected:Npn \cs_new_eq:NN #1
2051 {
2052   \__kernel_chk_if_free_cs:N #1
2053   \tex_global:D \cs_set_eq:NN #1
2054 }
2055 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
2056 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
2057 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End of definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 21.)

46.13 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting `TeX` conditionals in case `#1` is unbalanced in this matter. We optimize the case where the command exists by reducing as much as possible the tokens in the conditional.

```

2058 \cs_new_protected:Npn \cs_undefine:N #1
2059 { \cs_gset_eq:NN #1 \tex_undefined:D }
2060 \cs_new_protected:Npn \cs_undefine:c #1
2061 {
2062   \if_cs_exist:w #1 \cs_end:
2063   \else:
2064     \use_i:nnnn
2065   \fi:
2066   \exp_args:Nc \cs_undefine:N {#1}
2067 }

```

(End of definition for `\cs_undefine:N`. This function is documented on page 22.)

46.14 Generating parameter text from argument count

```

2068 <@@=cs>

```

```

\__kernel_cs_parm_from_arg_count:nnF
\__cs_parm_from_arg_count_test:nnF

```

`LaTeX3` provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

2069 \cs_new_protected:Npn \__kernel_cs_parm_from_arg_count:nnF #1#2
2070 {
2071   \exp_args:Ne \__cs_parm_from_arg_count_test:nnF
2072   {
2073     \exp_after:wN \exp_not:n
2074     \if_case:w \int_eval:n {#2}
2075     { }
2076     \or: { ##1 }
2077     \or: { ##1##2 }
2078     \or: { ##1##2##3 }
2079     \or: { ##1##2##3##4 }
2080     \or: { ##1##2##3##4##5 }
2081     \or: { ##1##2##3##4##5##6 }
2082     \or: { ##1##2##3##4##5##6##7 }
2083     \or: { ##1##2##3##4##5##6##7##8 }
2084     \or: { ##1##2##3##4##5##6##7##8##9 }
2085     \else: { \c_false_bool }
2086   \fi:
2087 }
2088 {#1}

```

```

2089 }
2090 \cs_new_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
2091 {
2092   \if_meaning:w \c_false_bool #1
2093     \exp_after:wN \use_ii:nn
2094   \else:
2095     \exp_after:wN \use_i:nn
2096   \fi:
2097   { #2 {#1} }
2098 }

```

(End of definition for `\__kernel_cs_parm_from_arg_count:nnF` and `\__cs_parm_from_arg_count_test:nnF`.)

46.15 Defining functions from a given number of arguments

2099 `<@@=cs>`

`\__cs_count_signature:N` Counting the number of tokens in the signature, i.e., the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

2100 \cs_new:Npn \__cs_count_signature:N #1
2101 { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2102 \cs_new:Npn \__cs_count_signature:n #1
2103 { \int_eval:n { \__cs_count_signature:nnN #1 } }
2104 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2105 {
2106   \if_meaning:w \c_true_bool #3
2107     \tl_count:n {#2}
2108   \else:
2109     -1
2110   \fi:
2111 }
2112 \cs_new:Npn \__cs_count_signature:c
2113 { \exp_args:Nc \__cs_count_signature:N }

```

(End of definition for `\__cs_count_signature:N`, `\__cs_count_signature:n`, and `\__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since T_EX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

2114 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
2115 {
2116   \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2117   {
2118     \msg_error:nnee { kernel } { bad-number-of-arguments }
2119     { \token_to_str:N #1 } { \int_eval:n {#3} }

```

```

2120         \use_none:n
2121     }
2122     {#4}
2123 }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

2124 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2125 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2126 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2127 { \exp_args:NNc \cs_generate_from_arg_count:NNnn }

```

(End of definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 21.)

46.16 Using the signature to define functions

```

2128 <@@=cs>

```

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn
\cs_set:Ne
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Ne
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Ne
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Ne
\cs_set_protected_nopar:Nx

```

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
  { \@@_count_signature:N #1 } {#2}
}

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

\cs_set:Npn \__cs_tmp:w #1#2#3
{
  \cs_new_protected:cpx { cs_ #1 : #2 }
  {
    \exp_not:N \__cs_generate_from_signature:NNn
    \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
  }
}
\cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
{
  \use:e
  {
    \__cs_generate_from_signature:nnNNnn
    \cs_split_function:N #2
  }
}
\cs_new:Nn
\cs_new:Ne
\cs_new:Nx
\cs_new_protected:Npn \__cs_generate_from_signature:nnNNnn #1#2#3#4#5#6
{
  \bool_if:NTF #3
  {
    \cs_set_nopar:Npx \__cs_tmp:w

```



```

2151         { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
2152 \tl_if_empty:oF \__cs_tmp:w
2153 {
2154     \msg_error:nneee { kernel } { non-base-function }
2155     { \token_to_str:N #5 } {#2} { \__cs_tmp:w }
2156 }
2157 \cs_generate_from_arg_count:NNnn
2158 #5 #4 { \tl_count:n {#2} } {#6}
2159 }
2160 {
2161     \msg_error:nne { kernel } { missing-colon }
2162     { \token_to_str:N #5 }
2163 }
2164 }
2165 \cs_new:Npn \__cs_generate_from_signature:n #1
2166 {
2167     \if:w n #1 \else: \if:w N #1 \else:
2168     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2169 }

```

Then we define the 24 variants beginning with N.

```

2170 \__cs_tmp:w { set } { Nn } { Npn }
2171 \__cs_tmp:w { set } { Ne } { Npe }
2172 \__cs_tmp:w { set } { Nx } { Npx }
2173 \__cs_tmp:w { set_nopar } { Nn } { Npn }
2174 \__cs_tmp:w { set_nopar } { Ne } { Npe }
2175 \__cs_tmp:w { set_nopar } { Nx } { Npx }
2176 \__cs_tmp:w { set_protected } { Nn } { Npn }
2177 \__cs_tmp:w { set_protected } { Ne } { Npe }
2178 \__cs_tmp:w { set_protected } { Nx } { Npx }
2179 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2180 \__cs_tmp:w { set_protected_nopar } { Ne } { Npe }
2181 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2182 \__cs_tmp:w { gset } { Nn } { Npn }
2183 \__cs_tmp:w { gset } { Ne } { Npe }
2184 \__cs_tmp:w { gset } { Nx } { Npx }
2185 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
2186 \__cs_tmp:w { gset_nopar } { Ne } { Npe }
2187 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
2188 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2189 \__cs_tmp:w { gset_protected } { Ne } { Npe }
2190 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2191 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2192 \__cs_tmp:w { gset_protected_nopar } { Ne } { Npe }
2193 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2194 \__cs_tmp:w { new } { Nn } { Npn }
2195 \__cs_tmp:w { new } { Ne } { Npe }
2196 \__cs_tmp:w { new } { Nx } { Npx }
2197 \__cs_tmp:w { new_nopar } { Nn } { Npn }
2198 \__cs_tmp:w { new_nopar } { Ne } { Npe }
2199 \__cs_tmp:w { new_nopar } { Nx } { Npx }
2200 \__cs_tmp:w { new_protected } { Nn } { Npn }
2201 \__cs_tmp:w { new_protected } { Ne } { Npe }
2202 \__cs_tmp:w { new_protected } { Nx } { Npx }
2203 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }

```

```

2204 \__cs_tmp:w { new_protected_nopar } { Ne } { Npe }
2205 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End of definition for \cs_set:Nn and others. These functions are documented on page 20.)

\cs_set:cn The 24 c variants simply use \exp_args:Nc.

\cs_set:ce 2206 \cs_set:Npn __cs_tmp:w #1#2

\cs_set:cx 2207 {

\cs_set_nopar:cn 2208 \cs_new_protected:cpx { cs_ #1 : c #2 }

\cs_set_nopar:ce 2209 {

\cs_set_nopar:cx 2210 \exp_not:N \exp_args:Nc

\cs_set_protected:cn 2211 \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:

\cs_set_protected:ce 2212 }

\cs_set_protected:cx 2213 }

\cs_set_protected_nopar:cn 2214 __cs_tmp:w { set } { n }

\cs_set_protected_nopar:ce 2215 __cs_tmp:w { set } { e }

\cs_set_protected_nopar:cx 2216 __cs_tmp:w { set } { x }

\cs_gset:cn 2217 __cs_tmp:w { set_nopar } { n }

\cs_gset:ce 2218 __cs_tmp:w { set_nopar } { e }

\cs_gset:cx 2219 __cs_tmp:w { set_nopar } { x }

\cs_gset_nopar:cn 2220 __cs_tmp:w { set_protected } { n }

\cs_gset_nopar:ce 2221 __cs_tmp:w { set_protected } { e }

\cs_gset_nopar:cx 2222 __cs_tmp:w { set_protected } { x }

\cs_gset_protected:cn 2223 __cs_tmp:w { set_protected_nopar } { n }

\cs_gset_protected:ce 2224 __cs_tmp:w { set_protected_nopar } { e }

\cs_gset_protected:cx 2225 __cs_tmp:w { set_protected_nopar } { x }

\cs_gset_protected_nopar:cn 2226 __cs_tmp:w { gset } { n }

\cs_gset_protected_nopar:ce 2227 __cs_tmp:w { gset } { e }

\cs_gset_protected_nopar:cx 2228 __cs_tmp:w { gset } { x }

\cs_new:cn 2229 __cs_tmp:w { gset_nopar } { n }

\cs_new:ce 2230 __cs_tmp:w { gset_nopar } { e }

\cs_new:cx 2231 __cs_tmp:w { gset_nopar } { x }

\cs_new_nopar:cn 2232 __cs_tmp:w { gset_protected } { n }

\cs_new_nopar:ce 2233 __cs_tmp:w { gset_protected } { e }

\cs_new_nopar:cx 2234 __cs_tmp:w { gset_protected } { x }

\cs_new_protected:cn 2235 __cs_tmp:w { gset_protected_nopar } { n }

\cs_new_protected:ce 2236 __cs_tmp:w { gset_protected_nopar } { e }

\cs_new_protected:cx 2237 __cs_tmp:w { gset_protected_nopar } { x }

\cs_new_protected_nopar:cn 2238 __cs_tmp:w { new } { n }

\cs_new_protected_nopar:ce 2239 __cs_tmp:w { new } { e }

\cs_new_protected_nopar:cx 2240 __cs_tmp:w { new } { x }

\cs_new_protected_nopar:ce 2241 __cs_tmp:w { new_nopar } { n }

\cs_new_protected_nopar:cx 2242 __cs_tmp:w { new_nopar } { e }

2243 __cs_tmp:w { new_nopar } { x }

2244 __cs_tmp:w { new_protected } { n }

2245 __cs_tmp:w { new_protected } { e }

2246 __cs_tmp:w { new_protected } { x }

2247 __cs_tmp:w { new_protected_nopar } { n }

2248 __cs_tmp:w { new_protected_nopar } { e }

2249 __cs_tmp:w { new_protected_nopar } { x }

(End of definition for \cs_set:Nn. This function is documented on page 20.)

46.17 Checking control sequence equality

```

\cs_if_eq_p:NN Check if two control sequences are identical.
\cs_if_eq_p:cN 2250 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2251 {
\cs_if_eq_p:cc 2252 \if_meaning:w #1#2
\cs_if_eq:NNTF 2253 \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2254 }
\cs_if_eq:NcTF 2255 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2256 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2257 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
\cs_if_eq:ccTF 2258 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
\cs_if_eq:p:Nc 2259 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2260 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_if_eq:NcT 2261 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
\cs_if_eq:NcF 2262 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
\cs_if_eq_p:cc 2263 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 2264 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_if_eq:ccT 2265 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
\cs_if_eq:ccF 2266 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End of definition for \cs_if_eq:NNTF. This function is documented on page 29.)

46.18 Diagnostic functions

```

2267 <@@=kernel>
\__kernel_chk_defined:NT Error if the variable #1 is not defined.
2268 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2269 {
2270 \cs_if_exist:NTF #1
2271 {#2}
2272 {
2273 \msg_error:nne { kernel } { variable-not-defined }
2274 { \token_to_str:N #1 }
2275 }
2276 }

```

(End of definition for __kernel_chk_defined:NT.)

Simply using the \showthe primitive does not allow for line-wrapping, so instead use \tl_show:n and \tl_log:n (defined in l3tl and that performs line-wrapping). This displays >~<variable>=<value>. We expand the value before-hand as otherwise some integers (such as \currentgrouplevel or \currentgrouptype) altered by the line-wrapping code would show wrong values.

```

\__kernel_register_show:N
\__kernel_register_show:c
\__kernel_register_log:N
\__kernel_register_log:c
\__kernel_register_show_aux:NN
\__kernel_register_show_aux:nN
2277 \cs_new_protected:Npn \__kernel_register_show:N
2278 { \__kernel_register_show_aux:NN \tl_show:n }
2279 \cs_new_protected:Npn \__kernel_register_show:c
2280 { \exp_args:Nc \__kernel_register_show:N }
2281 \cs_new_protected:Npn \__kernel_register_log:N
2282 { \__kernel_register_show_aux:NN \tl_log:n }
2283 \cs_new_protected:Npn \__kernel_register_log:c
2284 { \exp_args:Nc \__kernel_register_log:N }

```

```

2285 \cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
2286 {
2287   \__kernel_chk_defined:NT #2
2288   {
2289     \exp_args:No \__kernel_register_show_aux:nNN
2290     { \tex_the:D #2 } #2 #1
2291   }
2292 }
2293 \cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3
2294 { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End of definition for __kernel_register_show:N and others.)

\cs_show:N Some control sequences have a very long name or meaning. Thus, simply using TeX's
\cs_show:c primitive `\show` could lead to overlong lines. The output of this primitive is mimicked
\cs_log:N to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-
\cs_log:c wrapping. We must expand the meaning before passing it to the wrapping code as
__kernel_show:NN otherwise we would wrongly see the definitions that are in place there. To get correct
escape characters, set the `\escapechar` in a group; this also localizes the assignment
performed by e-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their
argument to a control sequence within a group to avoid showing `\relax` for undefined
control sequences.

```

2295 \cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
2296 \cs_new_protected:Npn \cs_show:c
2297 { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2298 \cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
2299 \cs_new_protected:Npn \cs_log:c
2300 { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2301 \cs_new_protected:Npn \__kernel_show:NN #1#2
2302 {
2303   \group_begin:
2304   \int_set:Nn \tex_escapechar:D { '\ }
2305   \exp_args:NNe
2306   \group_end:
2307   #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2308 }

```

(End of definition for `\cs_show:N`, `\cs_log:N`, and `\__kernel_show:NN`. These functions are documented on page 22.)

\group_show_list: Wrapper around `\showgroups`. Getting TeX to write to the log without interruption the
\group_log_list: run is done by altering the interaction mode.
__kernel_group_show:NN

```

2309 \cs_new_protected:Npn \group_show_list:
2310 { \__kernel_group_show:NN \use_none:n 1 }
2311 \cs_new_protected:Npn \group_log_list:
2312 { \__kernel_group_show:NN \int_gzero:N 0 }
2313 \cs_new_protected:Npn \__kernel_group_show:NN #1#2
2314 {
2315   \use:e
2316   {
2317     #1 \tex_interactionmode:D
2318     \int_set:Nn \tex_tracingonline:D {#2}
2319     \int_set:Nn \tex_errorcontextlines:D { -1 }
2320     \exp_not:N \exp_after:wN \scan_stop:

```

```

2321     \tex_showgroups:D
2322     \int_gset:Nn \tex_interactionmode:D
2323     { \int_use:N \tex_interactionmode:D }
2324     \int_set:Nn \tex_tracingonline:D
2325     { \int_use:N \tex_tracingonline:D }
2326     \int_set:Nn \tex_errorcontextlines:D
2327     { \int_use:N \tex_errorcontextlines:D }
2328   }
2329 }

```

(End of definition for `\group_show_list:`, `\group_log_list:`, and `\__kernel_group_show:NN`. These functions are documented on page 15.)

46.19 Decomposing a macro definition

```

2330 (@@=cs)

```

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the parameter specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

Since LuaMetaTeX's version of `\meaning` doesn't give the same output as the classical one, we are implementing the core of these macros in Lua instead.

```

2331 \use:e
2332 {
2333   \exp_not:n { \cs_new:Npn \__cs_prefix_arg_replacement:wN #1 }
2334   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__cs_stop #4 }
2335 }
2336 { #4 {#1} {#2} {#3} }
2337
2338 \cs_if_exist:NTF \__cs_macro_prefix_spec:N
2339 {
2340   \cs_new:Npn \cs_prefix_spec:N #1
2341   {
2342     \token_if_macro:NTF #1
2343     { \__cs_macro_prefix_spec:N #1 }
2344     { \scan_stop: }
2345   }
2346 }
2347 {
2348   \cs_new:Npn \cs_prefix_spec:N #1
2349   {
2350     \token_if_macro:NTF #1
2351     {
2352       \exp_after:wN \__cs_prefix_arg_replacement:wN
2353       \token_to_meaning:N #1 \s__cs_stop \use_i:nnn
2354     }
2355     { \scan_stop: }
2356   }
2357 }
2358 \cs_if_exist:NTF \__cs_macro_parameter_spec:N

```

```

2359 {
2360   \cs_new:Npn \cs_parameter_spec:N #1
2361   {
2362     \token_if_macro:NTF #1
2363     { \__cs_macro_parameter_spec:N #1 }
2364     { \scan_stop: }
2365   }
2366 }
2367 {
2368   \cs_new:Npn \cs_parameter_spec:N #1
2369   {
2370     \token_if_macro:NTF #1
2371     {
2372       \exp_after:wN \__cs_prefix_arg_replacement:wN
2373       \token_to_meaning:N #1 \s__cs_stop \use_iii:nnn
2374     }
2375     { \scan_stop: }
2376   }
2377 }
2378 \cs_if_exist:NTF \__cs_macro_replacement_spec:N
2379 {
2380   \cs_new:Npn \cs_replacement_spec:N #1
2381   {
2382     \token_if_macro:NTF #1
2383     { \__cs_macro_replacement_spec:N #1 }
2384     { \scan_stop: }
2385   }
2386 }
2387 {
2388   \cs_new:Npn \cs_replacement_spec:N #1
2389   {
2390     \token_if_macro:NTF #1
2391     {
2392       \exp_after:wN \__cs_prefix_arg_replacement:wN
2393       \token_to_meaning:N #1 \s__cs_stop \use_iii:nnn
2394     }
2395     { \scan_stop: }
2396   }
2397 }
2398 \code

```

This is a modified version of code suggested by Max Chernoff (<https://chat.stackexchange.com/transcript/message/67947633#67947633>). We aim to provide the classical $\text{\TeX}$ appearance but covering $\varepsilon\text{-TeX}$: due to changes in the engine, the only useful prefix here is `\protected` at present. (This may need to be revisited once the engine is more stable.) This does not cover tokens other than control sequences, so we only use it for the internal needed here.

```

2399 (*lua)
2400 if status.luatex_engine == 'luametatex' then
2401   local function scan_full_csname()
2402     local t = get_next()
2403     local csname = get_csname(t)
2404     return t.active and active_prefix .. csname or csname, t
2405   end

```

```

2406 luacmd('__cs_macro_prefix_spec:N', function()
2407     local token = get_next()
2408     if get_protected(token) then
2409         sprint(-2, "\\protected ")
2410     end
2411 end, 'global')
2412 luacmd('__cs_macro_parameter_spec:N', function()
2413     local csname, token = scan_full_csname(true)
2414     if token.parameters == 0 then return end
2415     sprint(-2, get_macro(csname, false, true))
2416 end, 'global')
2417 luacmd('__cs_macro_replacement_spec:N', function()
2418     local csname = scan_full_csname(true)
2419     sprint(-2, get_macro(csname, false, false))
2420 end, 'global')
2421 end
2422 </lua>
2423 <*code>

```

(End of definition for `\cs_prefix_spec:N` and others. These functions are documented on page 24.)

46.20 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

2424 \cs_new:Npn \prg_do_nothing: { }

```

(End of definition for `\prg_do_nothing:.` This function is documented on page 14.)

46.21 Breaking out of mapping functions

```

2425 <@@=prg>

```

`\prg_break_point:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when
`\prg_map_break:Nn` the user decides to break out. This is done by putting the code that must be performed as an argument of `\__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `\__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```

2426 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2427 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2428 {
2429     #5
2430     \if_meaning:w #1 #4
2431         \exp_after:wN \use_iii:nnn
2432     \fi:
2433     \prg_map_break:Nn #1 {#2}
2434 }

```

(End of definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 74.)

`\prg_break_point:` Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```

2435 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2436 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2437 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}

```

(End of definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 75.)

46.22 Starting a paragraph

`\mode_leave_vertical:` The approach here is different to that used by $\text{\LaTeX}2_\epsilon$ or plain $\text{\TeX}$, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses a protected macro, equivalent to the `\quitvmode` primitive. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the $\text{\LaTeX}2_\epsilon$ version, the availability of ϵ - $\text{\TeX}$ means using a mode test can be done at for example the start of an `\halign`.

```

2438 \cs_new_protected:Npn \mode_leave_vertical:
2439 {
2440   \if_mode_vertical:
2441     \exp_after:wN \tex_indent:D
2442   \fi:
2443 }

```

(End of definition for `\mode_leave_vertical:`. This function is documented on page 31.)

```

2444 \</code>

```


Chapter 47

l3expan implementation

```
2445 \def\exp_after:wN{*code}
2446 \def\exp_not:N{@@=exp}
```

`\l__exp_tmp_tl` The `\exp_` module has its private variable to temporarily store the result of `x`-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End of definition for `\l__exp_tmp_tl`.)

`\exp_after:wN` These are defined in `l3basics`, as they are needed “early”. This is just a reminder of that
`\exp_not:N` fact!
`\exp_not:n`

(End of definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 42.)

47.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 47.7. In section 47.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_tmp_tl` This scratch token list variable is defined in `l3basics`.

(End of definition for `\l__exp_tmp_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::<Z>` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`\_exp_arg_next:nnn` #1 is the result of an expansion step, #2 is the remaining argument manipulations and
`\_exp_arg_next:Nnn` #3 is the current result of the expansion chain. This auxiliary function moves #1 back
after #3 in the input stream and checks if any expansion is left to be done by calling
#2. In by far the most cases we need to add a set of braces to the result of an argument
manipulation so it is more effective to do it directly here. Actually, so far only the `c` of
the final argument manipulation variants does not require a set of braces.

```
2447 \cs_new:Npn \_exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2448 \cs_new:Npn \_exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(End of definition for `\_exp_arg_next:nnn` and `\_exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

```
2449 \cs_new:Npn \::: #1 {#1}
```

(End of definition for `\:::`. This function is documented on page 45.)

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
2450 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End of definition for `\::n`. This function is documented on page 45.)

`\::N` This function is used to skip an argument that consists of a single token and doesn't need
to be expanded.

```
2451 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End of definition for `\::N`. This function is documented on page 45.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn't
need to be expanded. It is not wrapped in braces in the result.

```
2452 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End of definition for `\::p`. This function is documented on page 45.)

`\::c` This function is used to skip an argument that is turned into a control sequence without
expansion.

```
2453 \cs_new:Npn \::c #1 \::: #2#3
2454 { \exp_after:wN \_exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End of definition for `\::c`. This function is documented on page 45.)

`\::o` This function is used to expand an argument once.

```
2455 \cs_new:Npn \::o #1 \::: #2#3
2456 { \exp_after:wN \_exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End of definition for `\::o`. This function is documented on page 45.)

`\::e` With the `\expanded` primitive available, just expand.

```
2457 \cs_new:Npn \::e #1 \::: #2#3
2458 { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
```

(End of definition for `\::e`. This function is documented on page 45.)

\::f This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, `f`-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only `TEX` has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `e` and `x` argument type.

```

2459 \cs_new:Npn \::f #1 \::: #2#3
2460 {
2461   \exp_after:wN \__exp_arg_next:nnn
2462   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2463   {#1} {#2}
2464 }
2465 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End of definition for `\::f` and `\exp_stop_f:`. These functions are documented on page 45.)

\::x This function is used to expand an argument fully. We build in the expansion of `\__exp_arg_next:nnn`.

```

2466 \cs_new_protected:Npn \::x #1 \::: #2#3
2467 {
2468   \cs_set_nopar:Npe \l__exp_tmp_tl
2469   { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2470   \l__exp_tmp_tl
2471 }
```

(End of definition for `\::x`. This function is documented on page 45.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, **`\::V`** `muskip`, or built-in `TEX` register. The `V` version expects a single token whereas `v` like `c` creates a `csname` from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```

2472 \cs_new:Npn \::V #1 \::: #2#3
2473 {
2474   \exp_after:wN \__exp_arg_next:nnn
2475   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2476   {#1} {#2}
2477 }
2478 \cs_new:Npn \::v #1 \::: #2#3
2479 {
2480   \exp_after:wN \__exp_arg_next:nnn
2481   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2482   {#1} {#2}
2483 }
```

(End of definition for `\::v` and `\::V`. These functions are documented on page 45.)

`\__exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in $\TeX$ register such as `\count`. For the $\TeX$ registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:.`

```

2484 \cs_new:Npn \__exp_eval_register:N #1
2485 {
2486   \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:.` In that case we throw an error. We could let $\TeX$ do it for us but that would result in the rather obscure

! You can't use '`\relax`' after `\the`.

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

2487   \if_meaning:w \scan_stop: #1
2488   \__exp_eval_error_msg:w
2489   \fi:

```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a $\TeX$ register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

2490   \else:
2491     \exp_after:wN \use_i_ii:nnn
2492   \fi:
2493   \exp_after:wN \exp_end: \tex_the:D #1
2494 }
2495 \cs_new:Npn \__exp_eval_register:c #1
2496 { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

```

```

2497 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2498 {
2499   \fi:
2500   \fi:
2501   \msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2502   \exp_end:
2503 }

```

(End of definition for `\__exp_eval_register:N` and `\__exp_eval_error_msg:w`.)

47.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In l3basics.

`\exp_args:cc`

(End of definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 38.)

`\exp_args:NNc`

`\exp_args:Ncc`

`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```
2504 \cs_new:Npn \exp_args:NNc #1#2#3
2505 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2506 \cs_new:Npn \exp_args:Ncc #1#2#3
2507 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2508 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2509 {
2510   \exp_after:wN #1
2511   \cs:w #2 \exp_after:wN \cs_end:
2512   \cs:w #3 \exp_after:wN \cs_end:
2513   \cs:w #4 \cs_end:
2514 }
```

(End of definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 39.)

`\exp_args:No`

`\exp_args:NNo`

`\exp_args:NNNo`

Those lovely runs of expansion!

```
2515 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2516 \cs_new:Npn \exp_args:NNo #1#2#3
2517 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2518 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2519 { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(End of definition for `\exp_args:No`, `\exp_args:NNo`, and `\exp_args:NNNo`. These functions are documented on page 38.)

`\exp_args:Ne`

When the `\expanded` primitive is available, use it.

```
2520 \cs_new:Npn \exp_args:Ne #1#2
2521 { \exp_after:wN #1 \tex_expanded:D { {#2} } }
```

(End of definition for `\exp_args:Ne`. This function is documented on page 38.)

`\exp_args:Nf`

`\exp_args:NV`

`\exp_args:Nv`

```
2522 \cs_new:Npn \exp_args:Nf #1#2
2523 { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2524 \cs_new:Npn \exp_args:Nv #1#2
2525 {
2526   \exp_after:wN #1 \exp_after:wN
2527   { \exp:w \__exp_eval_register:c {#2} }
2528 }
2529 \cs_new:Npn \exp_args:NV #1#2
2530 {
2531   \exp_after:wN #1 \exp_after:wN
2532   { \exp:w \__exp_eval_register:N #2 }
2533 }
```

(End of definition for `\exp_args:Nf`, `\exp_args:Nv`, and `\exp_args:Nv`. These functions are documented on page 38.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

2534 \cs_new:Npn \exp_args:NNV #1#2#3
2535 {
2536   \exp_after:wN #1
2537   \exp_after:wN #2
2538   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2539 }
2540 \cs_new:Npn \exp_args:NNv #1#2#3
2541 {
2542   \exp_after:wN #1
2543   \exp_after:wN #2
2544   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2545 }
2546 \cs_new:Npn \exp_args:NNe #1#2#3
2547 {
2548   \exp_after:wN #1
2549   \exp_after:wN #2
2550   \tex_expanded:D { {#3} }
2551 }
2552 \cs_new:Npn \exp_args:NNf #1#2#3
2553 {
2554   \exp_after:wN #1
2555   \exp_after:wN #2
2556   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2557 }
2558 \cs_new:Npn \exp_args:Nco #1#2#3
2559 {
2560   \exp_after:wN #1
2561   \cs:w #2 \exp_after:wN \cs_end:
2562   \exp_after:wN {#3}
2563 }
2564 \cs_new:Npn \exp_args:NcV #1#2#3
2565 {
2566   \exp_after:wN #1
2567   \cs:w #2 \exp_after:wN \cs_end:
2568   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2569 }
2570 \cs_new:Npn \exp_args:Ncv #1#2#3
2571 {
2572   \exp_after:wN #1
2573   \cs:w #2 \exp_after:wN \cs_end:
2574   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2575 }
2576 \cs_new:Npn \exp_args:Ncf #1#2#3
2577 {
2578   \exp_after:wN #1
2579   \cs:w #2 \exp_after:wN \cs_end:
2580   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2581 }
2582 \cs_new:Npn \exp_args:NVV #1#2#3
2583 {
2584   \exp_after:wN #1

```

```

2585 \exp_after:wN { \exp:w \exp_after:wN
2586   \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2587 \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2588 }

```

(End of definition for `\exp_args:NNV` and others. These functions are documented on page 39.)

`\exp_args:NNNV`
`\exp_args:NNNv`
`\exp_args:NNNe`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:Ncco`

A few more that we can hand-tune.

```

2589 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2590 {
2591   \exp_after:wN #1
2592   \exp_after:wN #2
2593   \exp_after:wN #3
2594   \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2595 }
2596 \cs_new:Npn \exp_args:NNNv #1#2#3#4
2597 {
2598   \exp_after:wN #1
2599   \exp_after:wN #2
2600   \exp_after:wN #3
2601   \exp_after:wN { \exp:w \__exp_eval_register:c {#4} }
2602 }
2603 \cs_new:Npn \exp_args:NNNe #1#2#3#4
2604 {
2605   \exp_after:wN #1
2606   \exp_after:wN #2
2607   \exp_after:wN #3
2608   \tex_expanded:D { {#4} }
2609 }
2610 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2611 {
2612   \exp_after:wN #1
2613   \cs:w #2 \exp_after:wN \cs_end:
2614   \exp_after:wN #3
2615   \cs:w #4 \cs_end:
2616 }
2617 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2618 {
2619   \exp_after:wN #1
2620   \cs:w #2 \exp_after:wN \cs_end:
2621   \exp_after:wN #3
2622   \exp_after:wN {#4}
2623 }
2624 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2625 {
2626   \exp_after:wN #1
2627   \cs:w #2 \exp_after:wN \cs_end:
2628   \cs:w #3 \exp_after:wN \cs_end:
2629   \exp_after:wN {#4}
2630 }

```

(End of definition for `\exp_args:NNNV` and others. These functions are documented on page 39.)

`\exp_args:Nx`

```

2631 \cs_new_protected:Npn \exp_args:Nx #1#2
2632 { \use:x { \exp_not:N #1 {#2} } }

```

(End of definition for `\exp_args:Nx`. This function is documented on page 38.)

47.3 Last-unbraced versions

```

\__exp_arg_last_unbraced:nn
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced

```

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

2633 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2634 \cs_new:Npn \::o_unbraced \::: #1#2
2635 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2636 \cs_new:Npn \::V_unbraced \::: #1#2
2637 {
2638   \exp_after:wN \__exp_arg_last_unbraced:nn
2639   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2640 }
2641 \cs_new:Npn \::v_unbraced \::: #1#2
2642 {
2643   \exp_after:wN \__exp_arg_last_unbraced:nn
2644   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2645 }
2646 \cs_new:Npn \::e_unbraced \::: #1#2
2647 { \tex_expanded:D { \exp_not:n {#1} #2 } }
2648 \cs_new:Npn \::f_unbraced \::: #1#2
2649 {
2650   \exp_after:wN \__exp_arg_last_unbraced:nn
2651   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2652 }
2653 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2654 {
2655   \cs_set_nopar:Npe \l__exp_tmp_tl { \exp_not:n {#1} #2 }
2656   \l__exp_tmp_tl
2657 }

```

(End of definition for `\__exp_arg_last_unbraced:nn` and others. These functions are documented on page 45.)

```

\exp_last_unbraced:No
\exp_last_unbraced:Nv
\exp_last_unbraced:Nv
\exp_last_unbraced:Ne
\exp_last_unbraced:Nf
\exp_last_unbraced:NNo
\exp_last_unbraced:NNv
\exp_last_unbraced:NNf
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNv
\exp_last_unbraced:NNNf
\exp_last_unbraced:Nno
\exp_last_unbraced:Nnf
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:NNNNo
\exp_last_unbraced:NNNNf
\exp_last_unbraced:Nx

```

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

2658 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2659 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2660 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2661 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2662 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2663 \cs_new:Npn \exp_last_unbraced:Ne #1#2
2664 { \exp_after:wN #1 \tex_expanded:D {#2} }
2665 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2666 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2667 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2668 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2669 \cs_new:Npn \exp_last_unbraced:NNv #1#2#3
2670 {

```



```

2671     \exp_after:wN #1
2672     \exp_after:wN #2
2673     \exp:w \__exp_eval_register:N #3
2674   }
2675 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
2676 {
2677     \exp_after:wN #1
2678     \exp_after:wN #2
2679     \exp:w \exp_end_continue_f:w #3
2680 }
2681 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2682 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2683 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2684 {
2685     \exp_after:wN #1
2686     \cs:w #2 \exp_after:wN \cs_end:
2687     \exp:w \__exp_eval_register:N #3
2688 }
2689 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2690 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2691 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2692 {
2693     \exp_after:wN #1
2694     \exp_after:wN #2
2695     \exp_after:wN #3
2696     \exp:w \__exp_eval_register:N #4
2697 }
2698 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
2699 {
2700     \exp_after:wN #1
2701     \exp_after:wN #2
2702     \exp_after:wN #3
2703     \exp:w \exp_end_continue_f:w #4
2704 }
2705 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
2706 \cs_new:Npn \exp_last_unbraced:Nnf { \::n \::f_unbraced \:: }
2707 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
2708 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
2709 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
2710 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2711 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2712 \cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
2713 {
2714     \exp_after:wN #1
2715     \exp_after:wN #2
2716     \exp_after:wN #3
2717     \exp_after:wN #4
2718     \exp:w \exp_end_continue_f:w #5
2719 }
2720 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End of definition for `\exp_last_unbraced:No` and others. These functions are documented on page 41.)

`\exp_last_two_unbraced:Noo` If #2 is a single token then this can be implemented as
`\__exp_last_two_unbraced:noN`

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2721 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2722 { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2723 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2724 { \exp_after:wN #3 #2 #1 }

```

(End of definition for `\exp_last_two_unbraced:Noo` and `\__exp_last_two_unbraced:noN`. This function is documented on page 41.)

47.4 Preventing expansion

`\__kernel_exp_not:w` At the kernel level, we need the primitive behavior to allow expansion *before* the brace group.

```

2725 \cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D

```

(End of definition for `\__kernel_exp_not:w`.)

`\exp_not:c` All these except `\exp_not:c` call the kernel-internal `\__kernel_exp_not:w` namely `\tex_unexpanded:D`.

```

2726 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2727 \cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
2728 \cs_new:Npn \exp_not:e #1
2729 { \__kernel_exp_not:w \tex_expanded:D { {#1} } }
2730 \cs_new:Npn \exp_not:f #1
2731 { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2732 \cs_new:Npn \exp_not:V #1
2733 {
2734   \__kernel_exp_not:w \exp_after:wN
2735   { \exp:w \__exp_eval_register:N #1 }
2736 }
2737 \cs_new:Npn \exp_not:v #1
2738 {
2739   \__kernel_exp_not:w \exp_after:wN
2740   { \exp:w \__exp_eval_register:c {#1} }
2741 }

```

(End of definition for `\exp_not:c` and others. These functions are documented on page 42.)

47.5 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrarily” many expansions we need a method to invoke TeX’s
`\exp_end:` expansion mechanism in such a way that (a) we are able to stop it in a controlled manner
`\exp_end_continue_f:w` and (b) the result of what triggered the expansion in the first place is null, i.e., that we
`\exp_end_continue_f:nw` do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact
the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number

turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`'s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an `f`-type expansion we provide the alphabetic constant `^^@` that also represents 0 but this time TeX's syntax for a `<number>` continues searching for an optional space (and it continues expansion doing that) — see TeXbook page 269 for details.

```

2742 \group_begin:
2743   \tex_catcode:D '\^^@ = 13
2744   \cs_new_protected:Npn \exp_end_continue_f:w { '^^@ }

```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```

2745   \if_cs_exist:N ^^@
2746   \else:
2747     \cs_new:Npn ^^@
2748       { \msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2749   \fi:

```

The same but grabbing an argument to remove spaces and braces.

```

2750   \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
2751 \group_end:

```

(End of definition for `\exp:w` and others. These functions are documented on page 44.)

47.6 Defining function variants

```

2752 <@@=cs>

```

`\s__cs_mark` Internal scan marks. No l3quark yet, so do things by hand.

```

\s__cs_stop
2753 \cs_new_eq:NN \s__cs_mark \scan_stop:
2754 \cs_new_eq:NN \s__cs_stop \scan_stop:

```

(End of definition for `\s__cs_mark` and `\s__cs_stop`.)

`\q__cs_recursion_stop` Internal recursion quarks. No l3quark yet, so do things by hand.

```

2755 \cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }

```

(End of definition for `\q__cs_recursion_stop`.)

`\__cs_use_none_delimit_by_s_stop:w` Internal scan marks.

```

\__cs_use_i_delimit_by_s_stop:nw
\__cs_use_none_delimit_by_q_recursion_stop:w
2756 \cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
2757 \cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
2758 \cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w
2759   #1 \q__cs_recursion_stop { }

```

(End of definition for `\__cs_use_none_delimit_by_s_stop:w`, `\__cs_use_i_delimit_by_s_stop:nw`, and `\__cs_use_none_delimit_by_q_recursion_stop:w`.)

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`
`\cs_generate_variant:cn` #2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `\__cs_tmp:w` as either `\cs_new:Npe` or `\cs_new_protected:Npe`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```

2760 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2761 {
2762   \__cs_generate_variant:N #1
2763   \use:e
2764   {
2765     \__cs_generate_variant:nnNN
2766     \cs_split_function:N #1
2767     \exp_not:N #1
2768     \tl_to_str:n {#2} ,
2769     \exp_not:N \scan_stop: ,
2770     \exp_not:N \q__cs_recursion_stop
2771   }
2772 }
2773 \cs_new_protected:Npn \cs_generate_variant:cn
2774 { \exp_args:Nc \cs_generate_variant:Nn }
```

(End of definition for `\cs_generate_variant:Nn`. This function is documented on page 35.)

`\__cs_generate_variant:N`
`\__cs_generate_variant:ww`
`\__cs_generate_variant:wwNw`

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be T_EX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long_`, `\protected_`, `\protected\long_`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and #3 is `\cs_new_protected:Npe`, otherwise it is `\cs_new:Npe`.

```

2775 \cs_new_protected:Npe \__cs_generate_variant:N #1
2776 {
2777   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2778   \exp_not:N \exp_not:N #1 #1
2779   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe
2780   \exp_not:N \else:
2781   \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2782   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2783   \s__cs_mark
```

```

2784     \s__cs_mark \cs_new_protected:Npe
2785     \tl_to_str:n { pr }
2786     \s__cs_mark \cs_new:Npe
2787     \s__cs_stop
2788     \exp_not:N \fi:
2789 }
2790 \exp_last_unbraced:NNNNo
2791 \cs_new_protected:Npn \__cs_generate_variant:ww
2792 #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2793 { \__cs_generate_variant:wwNw #1 }
2794 \exp_last_unbraced:NNNNo
2795 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2796 #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2797 { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End of definition for `\__cs_generate_variant:N`, `\__cs_generate_variant:ww`, and `\__cs_generate_variant:wwNw`.)

`\__cs_generate_variant:nnNN` #1 : Base name.
#2 : Base signature.
#3 : Boolean.
#4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```

2798 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2799 {
2800     \if_meaning:w \c_false_bool #3
2801     \msg_error:nne { kernel } { missing-colon }
2802     { \token_to_str:c {#1} }
2803     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2804     \fi:
2805     \__cs_generate_variant:Nnnw #4 {#1}{#2}
2806 }

```

(End of definition for `\__cs_generate_variant:nnNN`.)

`\__cs_generate_variant:Nnnw` #1 : Base function.
#2 : Base name.
#3 : Base signature.
#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a `cV` variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.

- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace o-expansion by x-expansion. More generally, we can only convert N to c, or convert n to V, v, o, e, f, or x.

All this boils down to a few rules. Only n and N-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to n except for N and p-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within e-expansion. The result is given to `\__cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \s__cs_mark <errors> \s__cs_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by $\text{\TeX}$ when fetching the last argument for `\__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `\__cs_generate_variant_loop_end:nwwwNNnn`.

```

2807 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2808 {
2809   \if_meaning:w \scan_stop: #4
2810     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2811   \fi:
2812   \use:e
2813   {
2814     \exp_not:N \__cs_generate_variant:wwNN
2815     \__cs_generate_variant_loop:nNwN { }
2816     #4
2817     \__cs_generate_variant_loop_end:nwwwNNnn
2818     \s__cs_mark
2819     #3 ~
2820     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2821     { }
2822     \s__cs_stop
2823     \exp_not:N #1 {#2} {#4}
2824   }
2825   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2826 }

```

(End of definition for `\__cs_generate_variant:Nnnw`.)

<code>__cs_generate_variant_loop:nNwN</code>	#1 :	Last few consecutive letters common between the base and variant (more precisely,
<code>__cs_generate_variant_loop_base:N</code>		<code>__cs_generate_variant_same:N <letter></code> for each letter).
<code>__cs_generate_variant_loop_same:w</code>	#2 :	Next variant letter.
<code>__cs_generate_variant_loop_end:nwwwNNnn</code>	#3 :	Remainder of variant form.
<code>__cs_generate_variant_loop_long:wNNnn</code>	#4 :	Next base letter.

The first argument is populated by `\__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is V, v, o, e, f, or x. Otherwise, call `\__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second

argument of `\__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling `\__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `\__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the `<base name>` as #7, the `<variant signature>` #8, the `<next base letter>` #1 and the part #3 of the base signature that wasn't read yet; and combines those into the `<new function>` to be defined.
- If the end of the base form is encountered first, #4 is `~{} \fi:` which ends the conditional (with an empty expansion), followed by `\__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `\__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (`n` or `N` to support the variant). In that case too an error is placed as the second argument of `\__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `\__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```

2827 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
2828 {
2829   \if:w #2 #4
2830     \exp_after:wN \__cs_generate_variant_loop_same:w
2831   \else:
2832     \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2833       \if:w 0
2834         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2835         \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2836         0
2837         \__cs_generate_variant_loop_special:NNwNNnn #4#2
2838       \else:
2839         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2840       \fi:
2841     \fi:
2842   \fi:
2843   #1
2844   \prg_do_nothing:
2845   #2
2846   \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
2847 }
2848 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2849 {
2850   \if:w c #1 N \else:
2851     \if:w o #1 n \else:
2852       \if:w V #1 n \else:
2853         \if:w v #1 n \else:

```

```

2854         \if:w f #1 n \else:
2855         \if:w e #1 n \else:
2856         \if:w x #1 n \else:
2857         \if:w n #1 n \else:
2858         \if:w N #1 N \else:
2859         \scan_stop:
2860         \fi:
2861         \fi:
2862         \fi:
2863         \fi:
2864         \fi:
2865         \fi:
2866         \fi:
2867         \fi:
2868         \fi:
2869     }
2870 \cs_new:Npn \__cs_generate_variant_loop_same:w
2871     #1 \prg_do_nothing: #2#3#4
2872     { #3 { #1 \__cs_generate_variant_same:N #2 } }
2873 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2874     #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
2875     {
2876     \scan_stop: \scan_stop: \fi:
2877     \s__cs_mark \s__cs_stop
2878     \exp_not:N #6
2879     \exp_not:c { #7 : #8 #1 #3 }
2880     }
2881 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
2882     {
2883     \exp_not:n
2884     {
2885     \s__cs_mark
2886     \msg_error:nnee { kernel } { variant-too-long }
2887     {#5} { \token_to_str:N #3 }
2888     \use_none:nnn
2889     \s__cs_stop
2890     #3
2891     #3
2892     }
2893     }
2894 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2895     #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
2896     {
2897     \fi: \fi: \fi:
2898     \exp_not:n
2899     {
2900     \s__cs_mark
2901     \msg_error:nneeee { kernel } { invalid-variant }
2902     {#7} { \token_to_str:N #5 } {#1} {#2}
2903     \use_none:nnn
2904     \s__cs_stop
2905     #5
2906     #5
2907     }

```



```

2908 }
2909 \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
2910 #1#2#3 \s__cs_stop #4#5#6#7
2911 {
2912   #3 \s__cs_stop #4 #5 {#6} {#7}
2913   \exp_not:n
2914   {
2915     \msg_error:nneeee
2916     { kernel } { deprecated-variant }
2917     {#7} { \token_to_str:N #5 } {#1} {#2}
2918   }
2919 }

```

(End of definition for __cs_generate_variant_loop:nNwN and others.)

__cs_generate_variant_same:N When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the n-type no-expansion, but the N and p types require a slightly different behavior with respect to braces. For V-type this function could output N to avoid adding useless braces but that is not a problem.

```

2920 \cs_new:Npn \__cs_generate_variant_same:N #1
2921 {
2922   \if:w N #1 #1 \else:
2923     \if:w p #1 #1 \else:
2924       \token_to_str:N n
2925     \if:w n #1 \else:
2926       \__cs_generate_variant_loop_special:NNwNNnn #1#1
2927     \fi:
2928   \fi:
2929   \fi:
2930 }

```

(End of definition for __cs_generate_variant_same:N.)

__cs_generate_variant:wwNN If the variant form has already been defined, log its existence (provided log-functions is active). Otherwise, make sure that the \exp_args:N #3 form is defined, and if it contains x, change __cs_tmp:w locally to \cs_new_protected:Npe. Then define the variant by combining the \exp_args:N #3 variant and the base function.

```

2931 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2932 #1 \s__cs_mark #2 \s__cs_stop #3#4
2933 {
2934   #2
2935   \cs_if_free:NT #4
2936   {
2937     \group_begin:
2938     \__cs_generate_internal_variant:n {#1}
2939     \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2940     \group_end:
2941   }
2942 }

```

(End of definition for __cs_generate_variant:wwNN.)

__cs_generate_internal_variant:n First test for the presence of x (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting __cs_tmp:w). Then

call `\__cs_generate_internal_variant:NNn` with arguments `\cs_new_protected:cpn \use:x` (for protected) or `\cs_new:cpn \tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\::` commands. Otherwise, call `\__cs_generate_internal_one_go:NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```

2943 \cs_new_protected:Npe \__cs_generate_internal_variant:n #1
2944 {
2945   \exp_not:N \__cs_generate_internal_variant:wwnNwn
2946   #1 \s__cs_mark
2947   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe }
2948   \cs_new_protected:cpn
2949   \use:x
2950   \token_to_str:N x \s__cs_mark
2951   { }
2952   \cs_new:cpn
2953   \exp_not:N \tex_expanded:D
2954   \s__cs_stop
2955   {#1}
2956 }
2957 \exp_last_unbraced:NNNNo
2958 \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwn #1
2959 { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
2960 {
2961   #3
2962   \cs_if_free:cT { exp_args:N #7 }
2963   { \__cs_generate_internal_variant:NNn #4 #5 {#7} }
2964 }
2965 \cs_set_protected:Npn \__cs_tmp:w #1
2966 {
2967   \cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
2968   {
2969     \if_catcode:w X \use_none:nnnnnnnn ##3
2970     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
2971     \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
2972     \prg_do_nothing: \prg_do_nothing: X
2973     \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
2974     \else:
2975       \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
2976     \fi:
2977     ##3
2978     \s__cs_mark
2979     {
2980       \use:e
2981       {
2982         ##1 { exp_args:N ##3 }
2983         { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
2984       }
2985     }
2986     #1
2987     \s__cs_mark
2988     { \exp_not:n { \__cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }

```

```

2989     \s__cs_stop
2990 }
2991 \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
2992   ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
2993 \cs_new_eq:NN \__cs_generate_internal_test:Nw
2994   \__cs_generate_internal_test_aux:w
2995 }
2996 \exp_args:No \__cs_tmp:w { \token_to_str:N p }
2997 \cs_new_protected:Npn \__cs_generate_internal_one_go:NNn #1#2#3
2998 {
2999   \__cs_generate_internal_loop:nwnnw
3000   { \exp_not:N ##1 } 1 . { } { }
3001   #3 { ? \__cs_generate_internal_end:w } X ;
3002   23456789 { ? \__cs_generate_internal_long:w } ;
3003   #1 #2 {#3}
3004 }
3005 \cs_new_protected:Npn \__cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
3006 {
3007   \use_none:n #5
3008   \use_none:n #7
3009   \cs_if_exist_use:cF { __cs_generate_internal_#5:NN }
3010   { \__cs_generate_internal_other:NN }
3011   #5 #7
3012   #7 .
3013   { #3 #1 } { #4 ## #2 }
3014   #6 ;
3015 }
3016 \cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
3017 { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
3018 \cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
3019 { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
3020 \cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
3021 { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
3022 \cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
3023 { \__cs_generate_internal_loop:nwnnw { {###2} } }
3024 \cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
3025 {
3026   \exp_args:No \__cs_generate_internal_loop:nwnnw
3027   {
3028     \exp_after:wN
3029     {
3030       \exp:w \exp_args:NNc \exp_after:wN \exp_end:
3031       { exp_not:#1 } {###2}
3032     }
3033   }
3034 }
3035 \cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
3036 { #6 { exp_args:N #8 } #3 { #7 {#2} } }
3037 \cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
3038 {
3039   \exp_args:Nx \__cs_generate_internal_long:nnnNNn
3040   { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
3041   {#4} {#5}
3042 }

```

```

3043 \cs_new:Npn \__cs_generate_internal_long:nnnNnn #1#2#3#4 ; ; #5#6#7
3044 { #5 { exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting \: #1 (not expanded further). We avoid tests by putting a trailing : \use_i:nn, which leaves \cs_end: and removes the looping macro. The colon is in fact also turned into \::: so that the required structure for \exp_args:N... commands is correctly terminated.

```

3045 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
3046 {
3047   \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3048   \__cs_generate_internal_variant_loop:n
3049 }

```

(End of definition for __cs_generate_internal_variant:n and __cs_generate_internal_variant_loop:n.)

\prg_generate_conditional_variant:Nnn

```

\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn
\__cs_generate_variant_check:nn

3050 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
3051 {
3052   \use:e
3053   {
3054     \__cs_generate_variant:nnNnn
3055     \cs_split_function:N #1
3056   }
3057 }
3058 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3059 {
3060   \if_meaning:w \c_false_bool #3
3061   \msg_error:nne { kernel } { missing-colon }
3062   { \token_to_str:c {#1} }
3063   \__cs_use_i_delimit_by_s_stop:nw
3064   \fi:
3065   \exp_after:wN \__cs_generate_variant:w
3066   \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3067   \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3068 }
3069 \cs_new_protected:Npn \__cs_generate_variant:w
3070 #1 , #2 \s__cs_mark #3#4#5
3071 {
3072   \if_meaning:w \scan_stop: #1 \scan_stop:
3073   \if_meaning:w \q__cs_nil #1 \q__cs_nil
3074   \use_i:nnn
3075   \fi:
3076   \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3077   \else:
3078   \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3079   { {#3} {#4} {#5} }
3080   {
3081     \msg_error:nnee
3082     { kernel } { conditional-form-unknown }
3083     {#1} { \token_to_str:c { #3 : #4 } }
3084   }
3085   \fi:
3086   \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3087 }

```

```

3088 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3089 { \__cs_generate_variant_check:nn { #1_p : #2 } }
3090 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3091 { \__cs_generate_variant_check:nn { #1 : #2 T } }
3092 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3093 { \__cs_generate_variant_check:nn { #1 : #2 F } }
3094 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3095 { \__cs_generate_variant_check:nn { #1 : #2 TF } }
3096 \cs_new_protected:Npn \__cs_generate_variant_check:nn #1#2
3097 {
3098     \cs_if_exist:ctF {#1}
3099     { \cs_generate_variant:cn {#1} {#2} }
3100     {
3101         \msg_error:nne
3102         { kernel } { conditional-base-undefined }
3103         { \token_to_str:c {#1} }
3104     }
3105 }

```

`\exp_args_generate:n` This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we just call the internal function.

```

3106 \cs_new_protected:Npn \exp_args_generate:n #1
3107 {
3108     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3109     {
3110         \str_map_inline:nn {##1}
3111         {
3112             \str_if_in:nnF { NnpcofeVvx } {####1}
3113             {
3114                 \msg_error:nnnn { kernel } { invalid-exp-args }
3115                 {####1} {#1}
3116                 \str_map_break:n { \use_none:nn }
3117             }
3118         }
3119         \__cs_generate_internal_variant:n {##1}
3120     }
3121 }

```

47.7 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

`\exp_args:Nnc` Here are the actual function definitions, using the helper functions above. The group is used because `\_cs_generate_internal_variant:n` redefines `\_cs_tmp:w` locally.

```
\exp_args:Nno
```

```
\exp_args:NnV
```

```
\exp_args:Nnv
```

```
\exp_args:Nne
```

```
\exp_args:Nnf
```

```
\exp_args:Noc
```

```
\exp_args:Noo
```

```
\exp args:Nof
```

```
\exp args:NVo
```

```
\exp args:Nfo
```

```
\exp args:Nff
```

```
\exp args:Nee
```

```
3122 \cs_set_protected:Npn \__cs_tmp:w #1
```

```

3123 {
3124   \group_begin:
3125   \exp_args:No \__cs_generate_internal_variant:n
3126   { \tl_to_str:n {#1} }
3127   \group_end:
3128 }
3129 \__cs_tmp:w { nc }
3130 \__cs_tmp:w { no }
3131 \__cs_tmp:w { nV }
3132 \__cs_tmp:w { nv }
3133 \__cs_tmp:w { ne }
3134 \__cs_tmp:w { nf }
3135 \__cs_tmp:w { oc }
3136 \__cs_tmp:w { oo }
3137 \__cs_tmp:w { of }
3138 \__cs_tmp:w { Vo }
3139 \__cs_tmp:w { fo }
3140 \__cs_tmp:w { ff }
3141 \__cs_tmp:w { ee }
3142 \__cs_tmp:w { Nx }
3143 \__cs_tmp:w { cx }
3144 \__cs_tmp:w { nx }
3145 \__cs_tmp:w { ox }
3146 \__cs_tmp:w { xo }
3147 \__cs_tmp:w { xx }

```

(End of definition for `\exp_args:Nnc` and others. These functions are documented on page 39.)

```

\exp_args:NNcf
\exp_args:NNno
\exp_args:NNnV
\exp_args:NNoo
\exp_args:NNVV
\exp_args:Ncno
\exp_args:NcnV
\exp_args:Ncoo
\exp_args:NcVV
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nnnf
\exp_args:Nnff
\exp_args:Nooo
\exp_args:Noof
\exp_args:Nffo
\exp_args:Neee
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

```

3148 \__cs_tmp:w { Ncf }
3149 \__cs_tmp:w { Nno }
3150 \__cs_tmp:w { NnV }
3151 \__cs_tmp:w { Noo }
3152 \__cs_tmp:w { NVV }
3153 \__cs_tmp:w { cno }
3154 \__cs_tmp:w { cnV }
3155 \__cs_tmp:w { coo }
3156 \__cs_tmp:w { cVV }
3157 \__cs_tmp:w { nnc }
3158 \__cs_tmp:w { nno }
3159 \__cs_tmp:w { nnf }
3160 \__cs_tmp:w { nff }
3161 \__cs_tmp:w { ooo }
3162 \__cs_tmp:w { oof }
3163 \__cs_tmp:w { ffo }
3164 \__cs_tmp:w { eee }
3165 \__cs_tmp:w { NNx }
3166 \__cs_tmp:w { Nnx }
3167 \__cs_tmp:w { Nox }
3168 \__cs_tmp:w { nnx }
3169 \__cs_tmp:w { nox }
3170 \__cs_tmp:w { ccx }
3171 \__cs_tmp:w { cnx }
3172 \__cs_tmp:w { oox }

```

(End of definition for `\exp_args:NNcf` and others. These functions are documented on page 40.)

47.8 Held-over variant generation

```
\cs_generate_from_arg_count:NNno  
\cs_replacement_spec:c  
  \debug_assert:nNe  
  \debug_assert:nne
```

A couple of variants that are from early functions.

```
3173 \cs_generate_variant:Nn \cs_generate_from_arg_count:NNnn { NNno }  
3174 \cs_generate_variant:Nn \cs_replacement_spec:N { c }  
3175 \cs_generate_variant:Nn \debug_assert:nNn { nNe }  
3176 \cs_generate_variant:Nn \debug_assert:nnn { nne }
```

(End of definition for `\cs_generate_from_arg_count:NNnn` and others. These functions are documented on page 21.)

```
3177 </code>
```

Chapter 48

l3sort implementation

```
3178 <*code>
3179 <@@=sort>

\__sort_sep:
3180 \cs_new_eq:NN \__sort_sep: \__kernel_int_sep:

(End of definition for \__sort_sep:.)
```

48.1 Variables

```
\g__sort_tmp_seq
\g__sort_tmp_tl
```

Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For seq and tl this is more efficient than using `\use:e` (or some `\exp_args:NNNe`) to smuggle the definition outside the group since TeX does not need to re-read tokens. For clist we don't gain anything since the result is converted from seq to clist anyways.

```
3181 \seq_new:N \g__sort_tmp_seq
3182 \tl_new:N \g__sort_tmp_tl

(End of definition for \g__sort_tmp_seq and \g__sort_tmp_tl.)
```

```
\l__sort_length_int
\l__sort_min_int
\l__sort_top_int
\l__sort_max_int
\l__sort_true_max_int
```

The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int - 1`. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `\__sort_compute_range:.` That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```
3183 \int_new:N \l__sort_length_int
3184 \int_new:N \l__sort_min_int
3185 \int_new:N \l__sort_top_int
3186 \int_new:N \l__sort_max_int
3187 \int_new:N \l__sort_true_max_int

(End of definition for \l__sort_length_int and others.)
```


`\l__sort_block_int` Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

```

3188 \int_new:N \l__sort_block_int

```

(End of definition for `\l__sort_block_int`.)

`\l__sort_begin_int` When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```

3189 \int_new:N \l__sort_begin_int
3190 \int_new:N \l__sort_end_int

```

(End of definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int` When merging two blocks (whose end-points are `beg` and `end`), *A* starts from the high end of the low block, and decreases until reaching `beg`. The index *B* starts from the top of the range and marks the register in which a sorted item should be put. Finally, *C* points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. *C* starts from the upper limit of that range.

```

3191 \int_new:N \l__sort_A_int
3192 \int_new:N \l__sort_B_int
3193 \int_new:N \l__sort_C_int

```

(End of definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

`\s__sort_mark` Internal scan marks.

```

\s__sort_stop 3194 \scan_new:N \s__sort_mark
3195 \scan_new:N \s__sort_stop

```

(End of definition for `\s__sort_mark` and `\s__sort_stop`.)

48.2 Finding available `\toks` registers

`\__sort_shrink_range:` After `\__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `\__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```

3196 \cs_new_protected:Npn \__sort_shrink_range:
3197 {
3198   \int_set:Nn \l__sort_A_int
3199     { \l__sort_true_max_int - \l__sort_min_int + 1 }
3200   \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
3201   \__sort_shrink_range_loop:
3202   \int_set:Nn \l__sort_max_int
3203   {
3204     \int_compare:nNnTF
3205       { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
3206     {

```

```

3207         \l__sort_min_int
3208         + ( \l__sort_A_int - 1 ) / 2
3209         + \l__sort_block_int / 4
3210         - 1
3211     }
3212     { \l__sort_true_max_int - \l__sort_block_int / 2 }
3213 }
3214 }
3215 \cs_new_protected:Npn \__sort_shrink_range_loop:
3216 {
3217     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
3218     \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
3219     \exp_after:wN \__sort_shrink_range_loop:
3220 \fi:
3221 }

```

(End of definition for `\__sort_shrink_range:` and `\__sort_shrink_range_loop:.`)

`\__sort_compute_range:` First find out what `\toks` have not yet been assigned. There are many cases. In $\text{\LaTeX} 2_{\epsilon}$ with no package, available `\toks` range from `\count15 + 1` to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e) $\text{\TeX}$, or when the package `etex` is loaded in $\text{\LaTeX} 2_{\epsilon}$, redefine `\__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The `elocalloc` package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as `etex` or `elocalloc` could be loaded.

In Con $\text{\TeX}$ t MkIV the range is from `\c_syst_last_allocated_toks + 1` to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks + 1` to `\c_max_register_int`. In all these cases, call `\__sort_shrink_range:.`

```

3222 \cs_new_protected:Npn \__sort_compute_range:
3223 {
3224     \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
3225     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3226     \__sort_shrink_range:
3227     \if_meaning:w \loctoks \tex_undefined:D \else:
3228     \if_meaning:w \loctoks \scan_stop: \else:
3229     \__sort_redefine_compute_range:
3230     \__sort_compute_range:
3231 \fi:
3232 \fi:
3233 }
3234 \cs_new_protected:Npn \__sort_redefine_compute_range:
3235 {
3236     \cs_if_exist:cTF { ver@elocalloc.sty }
3237     {
3238         \cs_gset_protected:Npn \__sort_compute_range:
3239         {
3240             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3241             \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
3242             \__sort_shrink_range:
3243         }
3244     }
3245 }

```

```

3246     \cs_gset_protected:Npn \__sort_compute_range:
3247     {
3248         \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3249         \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
3250         \__sort_shrink_range:
3251     }
3252 }
3253 }
3254 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
3255 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
3256 {
3257     \cs_if_exist:NT #1
3258     {
3259         \cs_gset_protected:Npn \__sort_compute_range:
3260         {
3261             \int_set:Nn \l__sort_min_int { #1 + 1 }
3262             \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3263             \__sort_shrink_range:
3264         }
3265     }
3266 }

```

(End of definition for `\__sort_compute_range:`, `\__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

48.3 Protected user commands

`\__sort_main:NNNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `\__sort_level:` calls `\__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_tmp_seq` for `seq` and `clist`. This is done by `\__sort_seq:NNNNn` and `\__sort_tl:NNn`.

```

3267 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
3268 {
3269     \__sort_disable_tokstoksdef:
3270     \__sort_compute_range:
3271     \int_set_eq:NN \l__sort_top_int \l__sort_min_int
3272     #1 #3
3273     {
3274         \if_int_compare:w \l__sort_top_int = \l__sort_max_int
3275             \__sort_too_long_error:NNw #2 #3
3276         \fi:
3277         \tex_toks:D \l__sort_top_int {##1}
3278         \int_incr:N \l__sort_top_int
3279     }
3280     \int_set:Nn \l__sort_length_int
3281     { \l__sort_top_int - \l__sort_min_int }
3282     \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
3283     \int_set:Nn \l__sort_block_int { 1 }
3284     \__sort_level:
3285 }

```

(End of definition for `\__sort_main:NNNn`.)

`\tl_sort:Nn` Call the main sorting function then unpack `\toks` registers outside the group into the
`\tl_sort:cn` target token list. The unpacking is done by `\__sort_tl_toks:w`; registers are numbered
`\tl_gsort:Nn` from `\l__sort_min_int` to `\l__sort_top_int - 1`. For expansion behavior we need
`\tl_gsort:cn` a couple of primitives. The `\tl_gclear:N` reduces memory usage. The `\prg_break_`
`\__sort_tl:NNn` `point:` is used by `\__sort_main:NNNn` when the list is too long.
`\__sort_tl_toks:w`

```

3286 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
3287 \cs_generate_variant:Nn \tl_sort:Nn { c }
3288 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
3289 \cs_generate_variant:Nn \tl_gsort:Nn { c }
3290 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
3291 {
3292   \group_begin:
3293     \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
3294     \__kernel_tl_gset:Nx \g__sort_tmp_tl
3295     { \__sort_tl_toks:w \l__sort_min_int \__sort_sep: }
3296   \group_end:
3297   #1 #2 \g__sort_tmp_tl
3298   \tl_gclear:N \g__sort_tmp_tl
3299   \prg_break_point:
3300 }
3301 \cs_new:Npn \__sort_tl_toks:w #1 \__sort_sep:
3302 {
3303   \if_int_compare:w #1 < \l__sort_top_int
3304   { \tex_the:D \tex_toks:D #1 }
3305   \exp_after:wN \__sort_tl_toks:w
3306   \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN \__sort_sep:
3307   \fi:
3308 }

```

(End of definition for `\tl_sort:Nn` and others. These functions are documented on page 127.)

`\seq_sort:Nn` Use the same general framework for seq and clist. Apply the general sorting code, then
`\seq_sort:cn` unpack `\toks` into `\g__sort_tmp_seq`. Outside the group copy or convert (for clist) the
`\seq_gsort:Nn` data to the target variable. The `\seq_gclear:N` reduces memory usage. The `\prg_`
`\seq_gsort:cn` `break_point:` is used by `\__sort_main:NNNn` when the list is too long.
`\clist_sort:Nn`
`\clist_sort:cn`
`\clist_gsort:Nn`
`\clist_gsort:cn`
`\__sort_seq:NNNNn`

```

3309 \cs_new_protected:Npn \seq_sort:Nn
3310 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
3311 \cs_generate_variant:Nn \seq_sort:Nn { c }
3312 \cs_new_protected:Npn \seq_gsort:Nn
3313 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
3314 \cs_generate_variant:Nn \seq_gsort:Nn { c }
3315 \cs_new_protected:Npn \clist_sort:Nn
3316 {
3317   \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3318   \clist_set_from_seq:NN
3319 }
3320 \cs_generate_variant:Nn \clist_sort:Nn { c }
3321 \cs_new_protected:Npn \clist_gsort:Nn
3322 {
3323   \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3324   \clist_gset_from_seq:NN
3325 }

```

```

3326 \cs_generate_variant:Nn \clist_gsort:Nn { c }
3327 \cs_new_protected:Npn \__sort_seq:NNNNn #1#2#3#4#5
3328 {
3329   \group_begin:
3330     \__sort_main:NNNn #1 #2 #4 {#5}
3331     \seq_gclear:N \g__sort_tmp_seq
3332     \int_step_inline:nnn
3333       \l__sort_min_int { \l__sort_top_int - 1 }
3334       {
3335         \seq_gput_right:Ne \g__sort_tmp_seq
3336         { \tex_the:D \tex_toks:D ##1 }
3337       }
3338     \group_end:
3339     #3 #4 \g__sort_tmp_seq
3340     \seq_gclear:N \g__sort_tmp_seq
3341     \prg_break_point:
3342   }

```

(End of definition for `\seq_sort:Nn` and others. These functions are documented on page 163.)

48.4 Merge sort

`\__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

3343 \cs_new_protected:Npn \__sort_level:
3344 {
3345   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
3346     \l__sort_end_int \l__sort_min_int
3347     \__sort_merge_blocks:
3348     \tex_advance:D \l__sort_block_int \l__sort_block_int
3349     \exp_after:wN \__sort_level:
3350   \fi:
3351 }

```

(End of definition for `\__sort_level:.`)

`\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in *A*, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift *end* upwards by one more block, but keeping it $\leq$ *top*. Copy this upper block of `\toks` registers in registers above *length*, indexed by *C*: this is covered by `\__sort_copy_block:.` Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:`, after shifting *A*, *B* and *C* so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

3352 \cs_new_protected:Npn \__sort_merge_blocks:
3353 {
3354   \l__sort_begin_int \l__sort_end_int

```

```

3355 \tex_advance:D \l__sort_end_int \l__sort_block_int
3356 \if_int_compare:w \l__sort_end_int < \l__sort_top_int
3357   \l__sort_A_int \l__sort_end_int
3358   \tex_advance:D \l__sort_end_int \l__sort_block_int
3359   \if_int_compare:w \l__sort_end_int > \l__sort_top_int
3360     \l__sort_end_int \l__sort_top_int
3361   \fi:
3362   \l__sort_B_int \l__sort_A_int
3363   \l__sort_C_int \l__sort_top_int
3364   \__sort_copy_block:
3365   \int_decr:N \l__sort_A_int
3366   \int_decr:N \l__sort_B_int
3367   \int_decr:N \l__sort_C_int
3368   \exp_after:wN \__sort_merge_blocks_aux:
3369   \exp_after:wN \__sort_merge_blocks:
3370 \fi:
3371 }

```

(End of definition for __sort_merge_blocks:.)

`\__sort_copy_block:` We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```

3372 \cs_new_protected:Npn \__sort_copy_block:
3373 {
3374   \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
3375   \int_incr:N \l__sort_C_int
3376   \int_incr:N \l__sort_B_int
3377   \if_int_compare:w \l__sort_B_int = \l__sort_end_int
3378     \use_i:nn
3379   \fi:
3380   \__sort_copy_block:
3381 }

```

(End of definition for __sort_copy_block:.)

`\__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int − 1` and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```

3382 \cs_new_protected:Npn \__sort_merge_blocks_aux:
3383 {
3384   \exp_after:wN \__sort_compare:nn \exp_after:wN
3385   { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
3386   \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
3387   \prg_do_nothing:
3388   \__sort_return_mark:w
3389   \__sort_return_mark:w
3390   \s__sort_mark
3391   \__sort_return_none_error:

```

3392 }

(End of definition for `\__sort_merge_blocks_aux:`.)

`\sort_return_same:` Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `\__sort_return_none_error:` is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `\__sort_return_same:w` (or its swapped analogue) followed by `\__sort_return_none_error:`. Finally if two or more are called, `\__sort_return_two_error:` ends up before any `\__sort_return_mark:w`, so that it produces an error.

```

3393 \cs_new_protected:Npn \sort_return_same:
3394   #1 \__sort_return_mark:w #2 \s__sort_mark
3395   {
3396     #1
3397     #2
3398     \__sort_return_two_error:
3399     \__sort_return_mark:w
3400     \s__sort_mark
3401     \__sort_return_same:w
3402   }
3403 \cs_new_protected:Npn \sort_return_swapped:
3404   #1 \__sort_return_mark:w #2 \s__sort_mark
3405   {
3406     #1
3407     #2
3408     \__sort_return_two_error:
3409     \__sort_return_mark:w
3410     \s__sort_mark
3411     \__sort_return_swapped:w
3412   }
3413 \cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
3414 \cs_new_protected:Npn \__sort_return_none_error:
3415   {
3416     \msg_error:nnee { sort } { return-none }
3417     { \tex_the:D \tex_toks:D \l__sort_A_int }
3418     { \tex_the:D \tex_toks:D \l__sort_C_int }
3419     \__sort_return_same:w \__sort_return_none_error:
3420   }
3421 \cs_new_protected:Npn \__sort_return_two_error:
3422   {
3423     \msg_error:nnee { sort } { return-two }
3424     { \tex_the:D \tex_toks:D \l__sort_A_int }
3425     { \tex_the:D \tex_toks:D \l__sort_C_int }
3426   }

```

(End of definition for `\sort_return_same:` and others. These functions are documented on page 47.)

`\__sort_return_same:w` If the comparison function returns `same`, then the second argument fed to `\__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers `B` and `C`, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

3427 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:

```

```

3428 {
3429   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3430   \int_decr:N \l__sort_B_int
3431   \int_decr:N \l__sort_C_int
3432   \if_int_compare:w \l__sort_C_int < \l__sort_top_int
3433     \use_i:nn
3434   \fi:
3435   \__sort_merge_blocks_aux:
3436 }

```

(End of definition for __sort_return_same:w.)

__sort_return_swapped:w If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register *A*. Then shift the pointers *A* and *B* to the left, and go for one more step for the merger, unless the left block was exhausted (*A* goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by *C*, are copied to the merger by `\__sort_merge_blocks_end:`.

```

3437 \cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
3438 {
3439   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
3440   \int_decr:N \l__sort_B_int
3441   \int_decr:N \l__sort_A_int
3442   \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
3443     \__sort_merge_blocks_end: \use_i:nn
3444   \fi:
3445   \__sort_merge_blocks_aux:
3446 }

```

(End of definition for __sort_return_swapped:w.)

__sort_merge_blocks_end: This function's task is to copy the `\toks` registers in the block indexed by *C* to the merger indexed by *B*. The end can equally be detected by checking when *B* reaches the threshold `begin`, or when *C* reaches `top`.

```

3447 \cs_new_protected:Npn \__sort_merge_blocks_end:
3448 {
3449   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3450   \int_decr:N \l__sort_B_int
3451   \int_decr:N \l__sort_C_int
3452   \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
3453     \use_i:nn
3454   \fi:
3455   \__sort_merge_blocks_end:
3456 }

```

(End of definition for __sort_merge_blocks_end:.)

48.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of `\__sort:nnNnn`). The arguments of `\__sort:nnNnn` are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of the list, call `\tl_sort:nN` on #1 and on #2, placing #4 in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
{
  \tl_if_blank:nF {#1}
  {
    \__sort:nnNnn { } { } #2
    #1 \q__sort_recursion_tail \q__sort_recursion_stop
  }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
  \quark_if_recursion_tail_stop_do:nn {#5}
  { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
  #3 {#4} {#5}
  { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
  { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `\__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `\__sort:nnNnn`. For this, the list is prepared by changing each $\langle item \rangle$ of the original token list into $\langle command \rangle \{ \langle item \rangle \}$, just like sequences are stored. We arrange things such that the $\langle command \rangle$ is the $\langle conditional \rangle$ provided by the user: the loop over the $\langle prepared\ tokens \rangle$ then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
  #6 { \langle pivot \rangle } {#7} \langle loop big \rangle \langle loop small \rangle
  \langle extra arguments \rangle
}
\__sort_loop:wNn ... \langle prepared tokens \rangle
\langle end-loop \rangle {} \s__sort_stop
```

In this example, which matches the structure of `\__sort_quick_split_i:NnnnnNn` and a few other functions below, the `\__sort_loop:wNn` auxiliary normally receives the user’s $\langle conditional \rangle$ as #6 and an $\langle item \rangle$ as #7. This is compared to the $\langle pivot \rangle$ (the argument #5, not shown here), and the $\langle conditional \rangle$ leaves the $\langle loop big \rangle$ or $\langle loop small \rangle$ auxiliary, which both have the same form as `\__sort_loop:wNn`, receiving the

next pair $\langle conditional \rangle \{ \langle item \rangle \}$ as #6 and #7. At the end, #6 is the $\langle end-loop \rangle$ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `\__sort:nnNn`, which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} {#2} { #3 {#1} } {#4}
}
```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs $\langle conditional \rangle \{ \langle item \rangle \}$, so we have to replace {#6} above by { #5 {#6} }, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark { \langle code \rangle }`, and expands to $\langle code \rangle \langle sorted\ list \rangle$. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \s__sort_mark
{
  \__sort_quick_split:NnNn #1 ... \s__sort_mark { \langle code \rangle }
  { \langle pivot \rangle }
}
```

Items which are larger than the $\langle pivot \rangle$ are sorted, then placed after code that sorts the smaller items, and after the (braced) $\langle pivot \rangle$.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:nnnnNn` of the last example, but aware of whether the list of $\langle conditional \rangle \{ \langle item \rangle \}$ read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the $\langle end-loop \rangle$ function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the $\langle end-loop \rangle$ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when `TeX` encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical T_EX's memory.

`\tl_sort:nN`

`\__sort_quick_prepare:Nnnn`
`\__sort_quick_prepare_end:NNNnw`
`\__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The `prepare` auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as #4. The scene is then set up for `\__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```

3457 \cs_new:Npn \tl_sort:nN #1#2
3458 {
3459   \exp_not:f
3460   {
3461     \tl_if_blank:nF {#1}
3462     {
3463       \__sort_quick_prepare:Nnnn #2 { } { }
3464       #1
3465       { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
3466       \s__sort_stop
3467     }
3468   }
3469 }
3470 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
3471 {
3472   \prg_break: #4 \prg_break_point:
3473   \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
3474 }
3475 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
3476 {
3477   \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
3478   \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
3479   \s__sort_mark \s__sort_stop
3480 }
3481 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}

```

(End of definition for `\tl_sort:nN` and others. This function is documented on page 127.)

`\__sort_quick_split:NnNn`

`\__sort_quick_only_i:NnnnnNn`
`\__sort_quick_only_ii:NnnnnNn`
`\__sort_quick_split_i:NnnnnNn`
`\__sort_quick_split_ii:NnnnnNn`

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a *function* #6, and an item #7. The *function* is the user's *conditional* except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the *pivot* and the *item*, and calls the `only_i` or `split_i` auxiliaries if the *item* is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to

work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form $\langle conditional \rangle \{ \langle item \rangle \}$, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's $\langle conditional \rangle$ rather than an ending function.

```

3482 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
3483 {
3484   #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
3485   \__sort_quick_only_i:NnnnnNn
3486   \__sort_quick_single_end:nnnwnw
3487   { #3 {#4} } { } { } {#2}
3488 }
3489 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
3490 {
3491   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3492   \__sort_quick_only_i:NnnnnNn
3493   \__sort_quick_only_i_end:nnnwnw
3494   { #6 {#7} } { #3 #2 } { } {#5}
3495 }
3496 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
3497 {
3498   #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
3499   \__sort_quick_split_i:NnnnnNn
3500   \__sort_quick_only_ii_end:nnnwnw
3501   { #6 {#7} } { } { #4 #2 } {#5}
3502 }
3503 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
3504 {
3505   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3506   \__sort_quick_split_i:NnnnnNn
3507   \__sort_quick_split_end:nnnwnw
3508   { #6 {#7} } { #3 #2 } {#4} {#5}
3509 }
3510 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
3511 {
3512   #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3513   \__sort_quick_split_i:NnnnnNn
3514   \__sort_quick_split_end:nnnwnw
3515   { #6 {#7} } {#3} { #4 #2 } {#5}
3516 }

```

(End of definition for `\__sort_quick_split:NnNn` and others.)

```

\__sort_quick_end:nnTFNn
  \__sort_quick_single_end:nnnwnw
  \__sort_quick_only_i_end:nnnwnw
  \__sort_quick_only_ii_end:nnnwnw
  \__sort_quick_split_end:nnnwnw

```

The `\__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a `true` and a `false` branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\s__sort_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\s__sort_stop` and place #6 back into the input stream. When the lists #1 and #2 are empty, the `single` auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before

the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

3517 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
3518 \cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3519 { #5 {#3} #6 \s__sort_stop }
3520 \cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3521 {
3522   \__sort_quick_split:NnNn #1
3523   \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3524   {#3}
3525   #6 \s__sort_stop
3526 }
3527 \cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3528 {
3529   \__sort_quick_split:NnNn #2
3530   \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
3531   #6 \s__sort_stop
3532 }
3533 \cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3534 {
3535   \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark
3536   {
3537     \__sort_quick_split:NnNn #1
3538     \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3539     {#3}
3540   }
3541   #6 \s__sort_stop
3542 }

```

(End of definition for __sort_quick_end:nnTFNn and others.)

48.6 Messages

__sort_error: Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with __sort_level: jumping to the break point. This error recovery won't work in a group.

```

3543 \cs_new_protected:Npn \__sort_error:
3544 {
3545   \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
3546   \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
3547   \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
3548 }

```

(End of definition for __sort_error:.)

__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or similar commands in their comparison code: the \toks registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no \toks allocator.

```

3549 \cs_new_protected:Npn \__sort_disable_toksdef:
3550 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
3551 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
3552 {
3553   \msg_error:nne { sort } { toksdef }
3554   { \token_to_str:N #1 }
3555   \__sort_error:
3556   \tex_toksdef:D #1
3557 }
3558 \msg_new:nnnn { sort } { toksdef }
3559 { Allocation~of~\iow_char:N\toks-registers-impossible~while~sorting. }
3560 {
3561   The~comparison~code~used~for~sorting~a~list~has~attempted~to~
3562   define~#1~as~a~new~\iow_char:N\toks-register~using~
3563   \iow_char:N\newtoks~
3564   or~a~similar~command.~The~list~will~not~be~sorted.
3565 }

```

(End of definition for __sort_disable_toksdef: and __sort_disabled_toksdef:n.)

__sort_too_long_error:NNw When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

3566 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
3567 {
3568   \fi:
3569   \msg_error:nneee { sort } { too-large }
3570   { \token_to_str:N #2 }
3571   { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
3572   { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
3573   #1 \__sort_error:
3574 }
3575 \msg_new:nnnn { sort } { too-large }
3576 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
3577 {
3578   TeX~has~#2~toks~registers~still~available:~
3579   this~only~allows~to~sort~with~up~to~#3~
3580   items.~The~list~will~not~be~sorted.
3581 }

```

(End of definition for __sort_too_long_error:NNw.)

```

3582 \msg_new:nnnn { sort } { return-none }
3583 { The~comparison~code~did~not~return. }
3584 {
3585   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
3586   did~not~call~
3587   \iow_char:N\sort_return_same: ~nor~
3588   \iow_char:N\sort_return_swapped: .~
3589   Exactly~one~of~these~should~be~called.
3590 }
3591 \msg_new:nnnn { sort } { return-two }
3592 { The~comparison~code~returned~multiple~times. }
3593 {
3594   When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
3595   \iow_char:N\sort_return_same: ~or~

```

```

3596     \iow_char:N\sort_return_swapped: ~multiple~times.~
3597     Exactly~one~of~these~should~be~called.
3598   }
3599   \prop_gput:Nnn \g_msg_module_name_prop { sort } { LaTeX }
3600   \prop_gput:Nnn \g_msg_module_type_prop { sort } { }
3601 \end{code}

```

Chapter 49

l3tl-analysis implementation

3602 $\langle @@=tl \rangle$

49.1 Internal functions

$\backslash s\_tl$ The format used to store token lists internally uses the scan mark $\backslash s\_tl$ as a delimiter.

(End of definition for $\backslash s\_tl$.)

49.2 Internal format

The task of the `l3tl-analysis` module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in `l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any $\langle token \rangle$ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find $\langle tokens \rangle$ which both `o`-expand and `e/x`-expand to the given $\langle token \rangle$. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$\langle tokens \rangle \backslash s\_tl \langle catcode \rangle \langle char\ code \rangle \backslash s\_tl$

The $\langle tokens \rangle$ `o`- and `e/x`-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for `l3str-convert`). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter $\backslash s\_tl$ may not appear unbraced in $\langle tokens \rangle$. This is not a problem because we are careful to wrap control sequences in braces (as an argument to $\backslash exp\_not:n$) when converting from a general token list to the internal format.

The current rule for converting a $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both `o`-expands and `e/x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl 0 -1 \s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl 1 <char code> \s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl 2 <char code> \s__tl`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__tl <hex catcode> <char code> \s__tl`.

In contrast, for `\peek_analysis_map_inline:n` we must allow for an input stream containing `\outer` macros, so that wrapping all control sequences in `\exp_not:n` is unsafe. Instead, we write the more elaborate `\__kernel_exp_not:w \exp_after:wN { \exp_not:N \cs }`. (On the other hand we make a better effort by avoiding `\exp_not:n` for characters other than active and macro parameters.)

3603 `<*code>`

49.3 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__tl` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an e-expansion.

3604 `\scan_new:N \s__tl`

(End of definition for \s__tl.)

`\l__tl_analysis_token` The tokens in the token list are probed with the TeX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

3605 `\cs_new_eq:NN \l__tl_analysis_token ?`

3606 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

(End of definition for \l__tl_analysis_token and \l__tl_analysis_char_token.)

`\l__tl_peek_code_tl` Holds some code to be run once the next token has been fully analyzed in `\peek_analysis_map_inline:n`.

3607 `\tl_new:N \l__tl_peek_code_tl`

(End of definition for \l__tl_peek_code_tl.)

`\c__tl_peek_catcodes_tl` A token list containing the character number 32 (space) with all possible category codes except 1 and 2 (begin-group and end-group). Why 32? Because some LuaTeX versions only allow creation of catcode 10 (space) tokens with this character code, so that we decided to make `\char_generate:nn` refuse to create such weird spaces as well. We do not include the macro parameter case (catcode 6) because it cannot be used as a macro delimiter.

3608 `\group_begin:`

3609 `\char_set_active_eq:NN \ \scan_stop:`

3610 `\tl_const:Ne \c__tl_peek_catcodes_tl`

```

3611 {
3612   \char_generate:nn { 32 } { 3 } 3
3613   \char_generate:nn { 32 } { 4 } 4
3614   \char_generate:nn { 32 } { 7 } 7
3615   \char_generate:nn { 32 } { 8 } 8
3616   \c_space_tl \token_to_str:N A
3617   \char_generate:nn { 32 } { 11 } \token_to_str:N B
3618   \char_generate:nn { 32 } { 12 } \token_to_str:N C
3619   \char_generate:nn { 32 } { 13 } \token_to_str:N D
3620 }
3621 \group_end:

```

(End of definition for \c__tl_peek_catcodes_tl.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

```
3622 \int_new:N \l__tl_analysis_normal_int
```

(End of definition for \l__tl_analysis_normal_int.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
3623 \int_new:N \l__tl_analysis_index_int
```

(End of definition for \l__tl_analysis_index_int.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
3624 \int_new:N \l__tl_analysis_nesting_int
```

(End of definition for \l__tl_analysis_nesting_int.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

```
3625 \int_new:N \l__tl_analysis_type_int
```

(End of definition for \l__tl_analysis_type_int.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

`<tokens> \s__tl <catcode> <char code> \s__tl`

```
3626 \tl_new:N \g__tl_analysis_result_tl
```

(End of definition for \g__tl_analysis_result_tl.)

`\__tl_analysis_extract_charcode:`
`\__tl_analysis_extract_charcode_aux:w` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘`<char>`’. LuaMetaTeX uses a different format for `\meaning`, so there is a little work to do: we extract the Unicode value this contains.

```

3627 \cs_new:Npn \__tl_analysis_extract_charcode:
3628 {
3629   \exp_after:wN \__tl_analysis_extract_charcode_aux:w
3630   \token_to_meaning:N \l__tl_analysis_token
3631 }
3632 \cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ }

```

```

3633 \bool_lazy_and:nnT
3634 { \cs_if_exist_p:N \tex_luatexversion:D }
3635 { \int_compare_p:nNn { \int_div_truncate:nn { \tex_luatexversion:D } { 100 } } > 1 }
3636 {
3637   \cs_gset:Npn \__tl_analysis_extract_charcode_aux:w #1 + #2 ~ ' #3 ' {"#2}
3638 }

```

(End of definition for `\__tl_analysis_extract_charcode:` and `\__tl_analysis_extract_charcode_aux:w`.)

```

\__tl_analysis_cs_space_count:NN
\__tl_analysis_cs_space_count:w
\__tl_analysis_cs_space_count_end:w

```

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as `\__tl_sep:-`delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```

3639 \cs_new:Npe \__tl_analysis_cs_space_count:NN #1 #2
3640 {
3641   \exp_not:N \exp_after:wN #1
3642   \exp_not:N \int_value:w \exp_not:N \int_eval:w 0
3643   \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_cs_space_count:w
3644   \exp_not:N \token_to_str:N #2
3645   \exp_not:N \fi: \exp_not:N \__tl_analysis_cs_space_count_end:w
3646   \exp_not:N \__tl_sep: \c_space_tl !
3647 }
3648 \cs_new:Npn \__tl_analysis_cs_space_count:w #1 ~
3649 {
3650   \if_false: #1 #1 \fi:
3651   + 1
3652   \__tl_analysis_cs_space_count:w
3653 }
3654 \cs_new:Npn \__tl_analysis_cs_space_count_end:w \__tl_sep: #1 \fi: #2 !
3655 {
3656   \exp_after:wN \__tl_sep: \int_value:w
3657   \str_count_ignore_spaces:n {#1} \__tl_sep:
3658 }

```

(End of definition for `\__tl_analysis_cs_space_count:NN`, `\__tl_analysis_cs_space_count:w`, and `\__tl_analysis_cs_space_count_end:w`.)

49.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

⟨token 1⟩ \s__tl ⟨catcode 1⟩ ⟨char code 1⟩ \s__tl
⟨token 2⟩ \s__tl ⟨catcode 2⟩ ⟨char code 2⟩ \s__tl
... ⟨token N⟩ \s__tl ⟨catcode N⟩ ⟨char code N⟩ \s__tl

```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by `TeX`. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an `e`-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for `TEX` when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`\__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `\__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

3659 \cs_new_protected:Npn \__tl_analysis:n #1
3660 {
3661   \group_begin:
3662     \group_align_safe_begin:
3663     \__tl_analysis_a:n {#1}
3664     \__tl_analysis_b:n {#1}
3665     \group_align_safe_end:
3666     \group_end:
3667 }
```

(End of definition for `\__tl_analysis:n`.)

49.5 Disabling active characters

`\__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For `pTEX` and `upTEX` we skip characters beyond [0, 255] because `\lccode` only allows those values.

```

3668 \group_begin:
3669   \char_set_catcode_active:N \^^@
3670   \cs_new_protected:Npn \__tl_analysis_disable:n #1
3671   {
```

```

3672     \tex_lccode:D 0 = #1 \exp_stop_f:
3673     \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3674   }
3675   \bool_lazy_or:nnT
3676   { \sys_if_engine_ptex_p: }
3677   { \sys_if_engine_uptex_p: }
3678   {
3679     \cs_gset_protected:Npn \__tl_analysis_disable:n #1
3680     {
3681       \if_int_compare:w 256 > #1 \exp_stop_f:
3682       \tex_lccode:D 0 = #1 \exp_stop_f:
3683       \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3684       \fi:
3685     }
3686   }
3687 \group_end:

```

(End of definition for __tl_analysis_disable:n.)

`\__tl_analysis_disable_char:N` Similar to `\__tl_analysis_disable:n`, but it receives a normal character token, tests if that token is active (by turning it into a space: the active space has been undefined at this point), and if so, disables it. Even if the character is active and set equal to a primitive conditional, nothing blows up. Again, in pTeX and upTeX we skip characters beyond [0,255], which cannot be active anyways.

```

3688 \group_begin:
3689   \char_set_catcode_active:N \^^@
3690   \cs_new_protected:Npn \__tl_analysis_disable_char:N #1
3691   {
3692     \tex_lccode:D '#1 = 32 \exp_stop_f:
3693     \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
3694     \tex_let:D #1 \tex_undefined:D
3695     \fi:
3696   }
3697   \bool_lazy_or:nnT
3698   { \sys_if_engine_ptex_p: }
3699   { \sys_if_engine_uptex_p: }
3700   {
3701     \cs_gset_protected:Npn \__tl_analysis_disable_char:N #1
3702     {
3703       \if_int_compare:w 256 > '#1 \exp_stop_f:
3704       \tex_lccode:D '#1 = 32 \exp_stop_f:
3705       \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
3706       \tex_let:D #1 \tex_undefined:D
3707       \fi:
3708       \fi:
3709     }
3710   }
3711 \group_end:

```

(End of definition for __tl_analysis_disable_char:N.)

49.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {\token}` is non-empty, because the escape character is printable.

`\__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches `-1` when we read the closing brace.

```

3712 \cs_new_protected:Npn \__tl_analysis_a:n #1
3713 {
3714   \__tl_analysis_disable:n { 32 }
3715   \int_set:Nn \tex_escapechar:D { 92 }
3716   \int_zero:N \l__tl_analysis_normal_int
3717   \int_zero:N \l__tl_analysis_index_int
3718   \int_zero:N \l__tl_analysis_nesting_int
3719   \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
3720   \int_decr:N \l__tl_analysis_index_int
3721 }
```

(End of definition for `\__tl_analysis_a:n`.)

```

\__tl_analysis_a_loop:w Read one character and check its type.
3722 \cs_new_protected:Npn \__tl_analysis_a_loop:w
3723 { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }

```

(End of definition for __tl_analysis_a_loop:w.)

__tl_analysis_a_type:w At this point, \l__tl_analysis_token holds the meaning of the following token. We store in \l__tl_analysis_type_int information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, −1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over \l__tl_analysis_token if it matches with one of the character tokens (hence is not a primitive conditional).

```

3724 \cs_new_protected:Npn \__tl_analysis_a_type:w
3725 {
3726   \l__tl_analysis_type_int =
3727   \if_meaning:w \l__tl_analysis_token \c_space_token
3728     0
3729   \else:
3730     \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
3731       1
3732     \else:
3733       \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
3734         - 1
3735       \else:
3736         2
3737     \fi:
3738   \fi:
3739   \fi:
3740   \exp_stop_f:
3741   \if_case:w \l__tl_analysis_type_int
3742     \exp_after:wN \__tl_analysis_a_space:w
3743   \or: \exp_after:wN \__tl_analysis_a_bgroup:w
3744   \or: \exp_after:wN \__tl_analysis_a_safe:N
3745   \else: \exp_after:wN \__tl_analysis_a_egroup:w
3746   \fi:
3747 }

```

(End of definition for __tl_analysis_a_type:w.)

__tl_analysis_a_space:w In this branch, the following token's meaning is a blank space. Apply \string to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as \l__tl_analysis_char_token the first character of the string representation then test it in __tl_analysis_a_space_test:w. Also, since __tl_analysis_a_store: expects the special token to be stored in the relevant \toks register, we do that. The extra \exp_not:n is unnecessary of course, but it makes

the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

3748 \cs_new_protected:Npn \__tl_analysis_a_space:w
3749 {
3750   \tex_afterassignment:D \__tl_analysis_a_space_test:w
3751   \exp_after:wN \cs_set_eq:NN
3752   \exp_after:wN \l__tl_analysis_char_token
3753   \token_to_str:N
3754 }
3755 \cs_new_protected:Npn \__tl_analysis_a_space_test:w
3756 {
3757   \if_meaning:w \l__tl_analysis_char_token \c_space_token
3758     \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
3759     \__tl_analysis_a_store:
3760   \else:
3761     \int_incr:N \l__tl_analysis_normal_int
3762   \fi:
3763   \__tl_analysis_a_loop:w
3764 }

```

(End of definition for __tl_analysis_a_space:w and __tl_analysis_a_space_test:w.)

```

\__tl_analysis_a_bgroup:w
\__tl_analysis_a_egroup:w
\__tl_analysis_a_group:nw
\__tl_analysis_a_group_aux:w
  \__tl_analysis_a_group_auxii:w
  \__tl_analysis_a_group_test:w

```

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need \l__tl_analysis_char_token to be a separate control sequence from \l__tl_analysis_token, to compare them.

```

3765 \group_begin:
3766   \char_set_catcode_group_begin:N \^^@ % {
3767   \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
3768     { \__tl_analysis_a_group:nw { \exp_after:wN \^^@ \if_false: } \fi: } }
3769   \char_set_catcode_group_end:N \^^@
3770   \cs_new_protected:Npn \__tl_analysis_a_egroup:w
3771     { \__tl_analysis_a_group:nw { \if_false: { \fi: \^^@ } } % }
3772 \group_end:
3773 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
3774 {
3775   \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
3776   \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
3777   \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
3778     \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
3779   \fi:
3780   \__tl_analysis_disable:n { \tex_lccode:D 0 }

```



```

3781 \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
3782 }
3783 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
3784 {
3785   \if_meaning:w \l__tl_analysis_token \tex_undefined:D
3786     \exp_after:wN \__tl_analysis_a_safe:N
3787   \else:
3788     \exp_after:wN \__tl_analysis_a_group_auxii:w
3789   \fi:
3790 }
3791 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
3792 {
3793   \tex_afterassignment:D \__tl_analysis_a_group_test:w
3794   \exp_after:wN \cs_set_eq:NN
3795   \exp_after:wN \l__tl_analysis_char_token
3796   \token_to_str:N
3797 }
3798 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
3799 {
3800   \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
3801     \__tl_analysis_a_store:
3802   \else:
3803     \int_incr:N \l__tl_analysis_normal_int
3804   \fi:
3805   \__tl_analysis_a_loop:w
3806 }

```

(End of definition for __tl_analysis_a_bgroup:w and others.)

`\__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special

token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

3807 \cs_new_protected:Npn \__tl_analysis_a_store:
3808 {
3809   \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
3810   \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
3811     \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
3812   \fi:
3813   \tex_skip:D \l__tl_analysis_index_int
3814     = \l__tl_analysis_normal_int sp
3815     plus \l__tl_analysis_type_int sp \scan_stop:
3816   \int_incr:N \l__tl_analysis_index_int
3817   \int_zero:N \l__tl_analysis_normal_int
3818   \if_int_compare:w \l__tl_analysis_nesting_int = - \c_one_int
3819     \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
3820   \fi:
3821 }

```

(End of definition for __tl_analysis_a_store:.)

`\__tl_analysis_a_safe:N`
`\__tl_analysis_a_cs:ww`

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

3822 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
3823 {
3824   \if_charcode:w
3825     \scan_stop:
3826     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
3827     \scan_stop:
3828     \exp_after:wN \use_i:nn
3829   \else:
3830     \exp_after:wN \use_ii:nn
3831   \fi:
3832   {
3833     \__tl_analysis_disable_char:N #1
3834     \int_incr:N \l__tl_analysis_normal_int
3835   }
3836   { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
3837   \__tl_analysis_a_loop:w
3838 }
3839 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1 \__tl_sep: #2 \__tl_sep:
3840 {
3841   \if_int_compare:w #1 > \c_zero_int
3842     \tex_skip:D \l__tl_analysis_index_int

```

```

3843         = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
3844         \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
3845     \else:
3846         \tex_advance:D
3847     \fi:
3848     \l__tl_analysis_normal_int #2 \exp_stop_f:
3849 }

```

(End of definition for `\__tl_analysis_a_safe:N` and `\__tl_analysis_a_cs:ww`.)

49.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`\__tl_analysis_b:n` Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

3850 \cs_new_protected:Npn \__tl_analysis_b:n #1
3851 {
3852     \__kernel_tl_gset:Nx \g__tl_analysis_result_tl
3853     {
3854         \__tl_analysis_b_loop:w 0 \__tl_sep: #1
3855         \prg_break_point:
3856     }
3857 }
3858 \cs_new:Npn \__tl_analysis_b_loop:w #1 \__tl_sep:
3859 {
3860     \exp_after:wN \__tl_analysis_b_normals:ww
3861     \int_value:w \tex_skip:D #1 \__tl_sep: #1 \__tl_sep:
3862 }

```

(End of definition for `\__tl_analysis_b:n` and `\__tl_analysis_b_loop:w`.)

`\__tl_analysis_b_normals:ww` The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n` `{⟨token⟩}` `\s__tl` in the input stream (after e-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because `#3` could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```

3863 \cs_new:Npn \__tl_analysis_b_normals:ww #1 \__tl_sep:
3864 {
3865     \if_int_compare:w #1 = \c_zero_int
3866         \__tl_analysis_b_special:w
3867     \fi:
3868     \__tl_analysis_b_normal:wwN #1 \__tl_sep:
3869 }
3870 \cs_new:Npn \__tl_analysis_b_normal:wwN #1 \__tl_sep: #2 \__tl_sep: #3
3871 {
3872     \exp_not:n { \exp_not:n { #3 } } \s__tl
3873     \if_charcode:w
3874         \scan_stop:

```

```

3875     \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
3876     \scan_stop:
3877     \exp_after:wN \__tl_analysis_b_char:Nn
3878     \exp_after:wN \__tl_analysis_b_char_aux:nww
3879   \else:
3880     \exp_after:wN \__tl_analysis_b_cs:Nww
3881   \fi:
3882   #3 #1 \__tl_sep: #2 \__tl_sep:
3883 }

```

(End of definition for __tl_analysis_b_normals:ww and __tl_analysis_b_normal:wwN.)

__tl_analysis_b_char:Nn This function is called here with arguments __tl_analysis_b_char_aux:nww and a normal character, while in the peek analysis code it is called with \use_none:n and possibly a space character, which is why the function has signature Nn. If the normal token we grab is a character, leave $\langle catcode \rangle \langle charcode \rangle$ followed by $\backslash s\_tl$ in the input stream, and call __tl_analysis_b_normals:ww with its first argument decremented.

```

3884 \cs_new:Npe \__tl_analysis_b_char:Nn #1#2
3885 {
3886   \exp_not:N \if_meaning:w #2 \exp_not:N \tex_undefined:D
3887   \token_to_str:N D \exp_not:N \else:
3888   \exp_not:N \if_catcode:w #2 \c_catcode_other_token
3889   \token_to_str:N C \exp_not:N \else:
3890   \exp_not:N \if_catcode:w #2 \c_catcode_letter_token
3891   \token_to_str:N B \exp_not:N \else:
3892   \exp_not:N \if_catcode:w #2 \c_math_toggle_token      3
3893   \exp_not:N \else:
3894   \exp_not:N \if_catcode:w #2 \c_alignment_token        4
3895   \exp_not:N \else:
3896   \exp_not:N \if_catcode:w #2 \c_math_superscript_token 7
3897   \exp_not:N \else:
3898   \exp_not:N \if_catcode:w #2 \c_math_subscript_token   8
3899   \exp_not:N \else:
3900   \exp_not:N \if_catcode:w #2 \c_space_token
3901   \token_to_str:N A \exp_not:N \else:
3902   6
3903   \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
3904   #1 {#2}
3905 }
3906 \cs_new:Npn \__tl_analysis_b_char_aux:nww #1
3907 {
3908   \int_value:w '#1 \s_tl
3909   \exp_after:wN \__tl_analysis_b_normals:ww
3910   \int_value:w \int_eval:w - 1 +
3911 }

```

(End of definition for __tl_analysis_b_char:Nn and __tl_analysis_b_char_aux:nww.)

__tl_analysis_b_cs:Nww If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by $\backslash s\_tl$, and call __tl_analysis_b_normals:ww with updated arguments.

```

3912 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
3913 {
3914   0 -1 \s_tl

```

```

3915     \_tl_analysis_cs_space_count:NN \_tl_analysis_b_cs_test:ww #1
3916   }
3917 \cs_new:Npn \_tl_analysis_b_cs_test:ww
3918   #1 \_tl_sep: #2 \_tl_sep: #3 \_tl_sep: #4 \_tl_sep:
3919   {
3920     \exp_after:wN \_tl_analysis_b_normals:ww
3921     \int_value:w \int_eval:w
3922     \if_int_compare:w #1 = \c_zero_int
3923       #3
3924     \else:
3925       \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
3926     \fi:
3927     - #2
3928     \exp_after:wN \_tl_sep:
3929     \int_value:w \int_eval:n { #4 + #1 } \_tl_sep:
3930   }

```

(End of definition for _tl_analysis_b_cs:Nww and _tl_analysis_b_cs_test:ww.)

```

\_tl_analysis_b_special:w
\_tl_analysis_b_special_char:wN
\_tl_analysis_b_special_space:w

```

Here, #1 is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the \toks register: when e/x-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call _tl_analysis_b_loop:w with the next index.

```

3931 \group_begin:
3932   \char_set_catcode_other:N A
3933   \cs_new:Npn \_tl_analysis_b_special:w
3934     \fi: \_tl_analysis_b_normal:wwN 0 \_tl_sep: #1 \_tl_sep:
3935   {
3936     \fi:
3937     \if_int_compare:w #1 = \l_tl_analysis_index_int
3938       \exp_after:wN \prg_break:
3939     \fi:
3940     \tex_the:D \tex_toks:D #1 \s_tl
3941     \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
3942       \token_to_str:N A
3943     \or: 1
3944     \or: 1
3945     \else: 2
3946     \fi:
3947     \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
3948       \exp_after:wN \_tl_analysis_b_special_char:wN \int_value:w
3949     \else:
3950       \exp_after:wN \_tl_analysis_b_special_space:w \int_value:w
3951     \fi:
3952     \int_eval:n { 1 + #1 } \exp_after:wN \_tl_sep:
3953     \token_to_str:N
3954   }
3955 \group_end:
3956 \cs_new:Npn \_tl_analysis_b_special_char:wN #1 \_tl_sep: #2
3957   {
3958     \int_value:w '#2 \s_tl
3959     \_tl_analysis_b_loop:w #1 \_tl_sep:
3960   }

```

```

3961 \use:e
3962 {
3963   \cs_new:Npn \exp_not:N \__tl_analysis_b_special_space:w
3964     #1 \exp_not:N \__tl_sep: \c_space_tl
3965 }
3966 {
3967   32 \s__tl
3968   \__tl_analysis_b_loop:w #1 \__tl_sep:
3969 }

```

(End of definition for `\__tl_analysis_b_special:w`, `\__tl_analysis_b_special_char:wN`, and `\__tl_analysis_b_special_space:w`.)

49.8 Mapping through the analysis

```

\tl_analysis_map_inline:Nn
\tl_analysis_map_inline:nn
  \__tl_analysis_map:Nn
  \__tl_analysis_map:NwNw

```

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the payload macro, which runs the user code and has a name specific to that nesting depth. The looping macro grabs the `<tokens>`, `<catcode>` and `<char code>`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then calls the payload macro with the arguments in the correct order (this is the reason why we cannot directly use the same macro for looping and payload), and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

3970 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
3971 { \exp_args:No \tl_analysis_map_inline:nn #1 }
3972 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
3973 {
3974   \__tl_analysis:n {#1}
3975   \int_gincr:N \g__kernel_prg_map_int
3976   \exp_args:Nc \__tl_analysis_map:Nn
3977     { \__tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
3978 }
3979 \cs_new_protected:Npn \__tl_analysis_map:Nn #1#2
3980 {
3981   \cs_gset_protected:Npn #1 ##1##2##3 {#2}
3982   \exp_after:wN \__tl_analysis_map:NwNw \exp_after:wN #1
3983   \g__tl_analysis_result_tl
3984   \s__tl { ? \tl_map_break: } \s__tl
3985   \prg_break_point:Nn \tl_map_break:
3986     { \int_gdecr:N \g__kernel_prg_map_int }
3987 }
3988 \cs_new_protected:Npn \__tl_analysis_map:NwNw #1 #2 \s__tl #3 #4 \s__tl
3989 {
3990   \use_none:n #3
3991   #1 {#2} {#4} {#3}
3992   \__tl_analysis_map:NwNw #1
3993 }

```

(End of definition for `\tl_analysis_map_inline:Nn` and others. These functions are documented on page 48.)

49.9 Showing the results

`\tl_analysis_show:N` Add to `\__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```

3994 \cs_new_protected:Npn \tl_analysis_show:N
3995   { \__tl_analysis_show:NnnnN \msg_show:nneeee \tl_show:N {} {} }
3996 \cs_new_protected:Npn \tl_analysis_log:N
3997   {
3998     \__tl_analysis_show:NnnnN \msg_log:nneeee \tl_log:N
3999     { \iow_newline: >~ . } { . }
4000   }
4001 \cs_new_protected:Npn \__tl_analysis_show:NnnnN #1#2#3#4#5
4002   {
4003     \tl_if_exist:NTF #5
4004     {
4005       \exp_args:No \__tl_analysis:n {#5}
4006       #1 { tl } { show-analysis }
4007       { \token_to_str:N #5 } { \__tl_analysis_show: } {#3} {#4}
4008     }
4009     { #2 #3 }
4010   }

```

(End of definition for `\tl_analysis_show:N`, `\tl_analysis_log:N`, and `\__tl_analysis_show:NnnnN`. These functions are documented on page 48.)

`\tl_analysis_show:n` No existence test needed here.

`\tl_analysis_log:n`

```

4011 \cs_new_protected:Npn \tl_analysis_show:n
4012   { \__tl_analysis_show:NnnnN \msg_show:nneeee {} {} }
4013 \cs_new_protected:Npn \tl_analysis_log:n
4014   { \__tl_analysis_show:NnnnN \msg_log:nneeee { \iow_newline: >~ . } { . } }
4015 \cs_new_protected:Npn \__tl_analysis_show:NnnnN #1#2#3#4
4016   {
4017     \__tl_analysis:n {#4}
4018     #1 { tl } { show-analysis } { } { \__tl_analysis_show: } {#2} {#3}
4019   }

```

(End of definition for `\tl_analysis_show:n`, `\tl_analysis_log:n`, and `\__tl_analysis_show:NnnnN`. These functions are documented on page 48.)

`\__tl_analysis_show:` Here, #1 o- and e/x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

4020 \cs_new:Npn \__tl_analysis_show:
4021   {
4022     \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
4023     \s__tl { ? \prg_break: } \s__tl
4024     \prg_break_point:
4025   }
4026 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
4027   {
4028     \use_none:n #2
4029     \iow_newline: > \use:nn { ~ } { ~ }

```

```

4030 \if_int_compare:w "#2 = \c_zero_int
4031 \exp_after:wN \__tl_analysis_show_cs:n
4032 \else:
4033 \if_int_compare:w "#2 = 13 \exp_stop_f:
4034 \exp_after:wN \exp_after:wN
4035 \exp_after:wN \__tl_analysis_show_active:n
4036 \else:
4037 \exp_after:wN \exp_after:wN
4038 \exp_after:wN \__tl_analysis_show_normal:n
4039 \fi:
4040 \fi:
4041 {#1}
4042 \__tl_analysis_show_loop:wNw
4043 }

```

(End of definition for __tl_analysis_show: and __tl_analysis_show_loop:wNw.)

__tl_analysis_show_normal:n Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up T_EX's alignment status.

```

4044 \cs_new:Npn \__tl_analysis_show_normal:n #1
4045 {
4046 \exp_after:wN \token_to_str:N #1 ~
4047 ( \exp_after:wN \token_to_meaning:N #1 )
4048 }

```

(End of definition for __tl_analysis_show_normal:n.)

__tl_analysis_show_value:N This expands to the value of #1 if it has any.

```

4049 \cs_new:Npn \__tl_analysis_show_value:N #1
4050 {
4051 \token_if_expandable:NF #1
4052 {
4053 \token_if_chardef:NTF #1 \prg_break: { }
4054 \token_if_mathchardef:NTF #1 \prg_break: { }
4055 \token_if_dim_register:NTF #1 \prg_break: { }
4056 \token_if_int_register:NTF #1 \prg_break: { }
4057 \token_if_skip_register:NTF #1 \prg_break: { }
4058 \token_if_toks_register:NTF #1 \prg_break: { }
4059 \use_none:nnn
4060 \prg_break_point:
4061 \use:n { \exp_after:wN = \tex_the:D #1 }
4062 }
4063 }

```

(End of definition for __tl_analysis_show_value:N.)

__tl_analysis_show_cs:n Control sequences and active characters are printed in the same way, making sure not to go beyond the \l_iow_line_count_int. In case of an overflow, we replace the last characters by \c__tl_analysis_show_etc_str.

```

\__tl_analysis_show_active:n
\__tl_analysis_show_long:nn
\__tl_analysis_show_long_aux:nnnn
4064 \cs_new:Npn \__tl_analysis_show_cs:n #1
4065 { \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
4066 \cs_new:Npn \__tl_analysis_show_active:n #1
4067 { \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }

```



```

4068 \cs_new:Npn \__tl_analysis_show_long:nn #1
4069 {
4070   \__tl_analysis_show_long_aux:oofn
4071   { \token_to_str:N #1 }
4072   { \token_to_meaning:N #1 }
4073   { \__tl_analysis_show_value:N #1 }
4074 }
4075 \cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
4076 {
4077   \int_compare:nNnTF
4078   { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
4079   > { \l_iow_line_count_int - 3 }
4080   {
4081     \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
4082     {
4083       \l_iow_line_count_int - 3
4084       - \str_count:N \c__tl_analysis_show_etc_str
4085     }
4086     \c__tl_analysis_show_etc_str
4087   }
4088   { #1 ~ ( #4 #2 #3 ) }
4089 }
4090 \cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

```

(End of definition for `\__tl_analysis_show_cs:n` and others.)

49.10 Peeking ahead

`\peek_analysis_map_break:`
`\peek_analysis_map_break:n`

The break statements use the general `\prg_map_break:Nn`.

```

4091 \cs_new:Npn \peek_analysis_map_break:
4092 { \prg_map_break:Nn \peek_analysis_map_break: { } }
4093 \cs_new:Npn \peek_analysis_map_break:n
4094 { \prg_map_break:Nn \peek_analysis_map_break: }

```

(End of definition for `\peek_analysis_map_break:` and `\peek_analysis_map_break:n`. These functions are documented on page [214](#).)

`\l__tl_peek_charcode_int`

```

4095 \int_new:N \l__tl_peek_charcode_int

```

(End of definition for `\l__tl_peek_charcode_int`.)

`\__tl_analysis_char_arg:Nw`
`\__tl_analysis_char_arg_aux:Nw`

After a call to `\futurelet \l__tl_analysis_token` followed by a stringified character token (either explicit space or catcode other character), grab the argument and pass it to #1. We only need to do anything in the case of a space.

```

4096 \cs_new:Npn \__tl_analysis_char_arg:Nw
4097 {
4098   \if_meaning:w \l__tl_analysis_token \c_space_token
4099   \exp_after:wN \__tl_analysis_char_arg_aux:Nw
4100   \fi:
4101 }
4102 \cs_new:Npn \__tl_analysis_char_arg_aux:Nw #1 ~ { #1 { ~ } }

```

(End of definition for `\__tl_analysis_char_arg:Nw` and `\__tl_analysis_char_arg_aux:Nw`.)

```

\peek_analysis_map_inline:n
\__tl_peek_analysis_loop:NNn
  \__tl_peek_analysis_test:
  \__tl_peek_analysis_exp:N
    \__tl_peek_analysis_exp_aux:N
\__tl_peek_analysis_nonexp:N
  \__tl_peek_analysis_cs:N
  \__tl_peek_analysis_char:N
  \__tl_peek_analysis_char:w
\__tl_peek_analysis_special:
\__tl_peek_analysis_retest:
  \__tl_peek_analysis_str:
  \__tl_peek_analysis_str:w
  \__tl_peek_analysis_str:n
    \__tl_peek_analysis_active_str:n
    \__tl_peek_analysis_explicit:n
\__tl_peek_analysis_escape:
  \__tl_peek_analysis_collect:w
  \__tl_peek_analysis_collect:n
  \__tl_peek_analysis_collect_loop:
  \__tl_peek_analysis_collect_test:
  \__tl_peek_analysis_collect_end:NNNN

```

Save the user's code in a control sequence that is suitable for nested maps. We may wish to pass to this function an `\outer` control sequence or active character; for this we will undefine any expandable token (testing if it is `\outer` is much slower) within a group, closed immediately after the function reads its arguments to avoid affecting the user's code or even our peek code (there is no risk of undefining `\group_end:` itself since that is not expandable). This user's code function also calls the loop auxiliary, and includes the trailing `\prg_break_point:Nn` for when the user wants to stop the loop. The loop auxiliary must remove that break point because it must look at the input stream.

```

4103 \cs_new_protected:Npn \peek_analysis_map_inline:n #1
4104 {
4105   \group_align_safe_begin:
4106   \int_gincr:N \g__kernel_prg_map_int
4107   \cs_set_protected:cpn
4108     { \__tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4109     ##1##2##3
4110   {
4111     \group_end:
4112     #1
4113     \__tl_peek_analysis_loop:NNn
4114     \prg_break_point:Nn \peek_analysis_map_break:
4115     {
4116       \int_gdecr:N \g__kernel_prg_map_int
4117       \group_align_safe_end:
4118     }
4119   }
4120   \__tl_peek_analysis_loop:NNn ? ? ?
4121 }

```

The loop starts a group (closed by the user-code function defined above) with a normalized escape character, and checks if the next token is special or N-type (distinguishing expandable from non-expandable tokens). The test for nonexpandable tokens in `\__tl_peek_analysis_test:` must be done after the tests for begin-group, end-group, and space tokens, in case `\l_peek_token` is either `\outer` or is a primitive T_EX conditional, as such tokens cannot be skipped over correctly by conditional code.

```

4122 \cs_new_protected:Npn \__tl_peek_analysis_loop:NNn #1#2#3
4123 {
4124   \group_begin:
4125   \tl_set:Nc \l__tl_peek_code_tl
4126     {
4127       \exp_not:c
4128       { \__tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4129     }
4130   \int_set:Nn \tex_escapechar:D { '\ }
4131   \peek_after:Nw \__tl_peek_analysis_test:
4132 }
4133 \cs_new_protected:Npn \__tl_peek_analysis_test:
4134 {
4135   \if_case:w
4136     \if_catcode:w \exp_not:N \l_peek_token { \c_max_int \fi:
4137     \if_catcode:w \exp_not:N \l_peek_token } \c_max_int \fi:
4138     \if_meaning:w \l_peek_token \c_space_token \c_max_int \fi:
4139     \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
4140     \c_one_int

```

```

4141     \fi:
4142     \c_zero_int
4143     \exp_after:wN \exp_after:wN
4144     \exp_after:wN \_tl_peek_analysis_exp:N
4145     \exp_after:wN \exp_not:N
4146   \or:
4147     \exp_after:wN \_tl_peek_analysis_nonexp:N
4148   \else:
4149     \exp_after:wN \_tl_peek_analysis_special:
4150   \fi:
4151 }

```

Expandable tokens (which are automatically N-type) can be `\outer` macros, hence the need for `\exp_after:wN` and `\exp_not:N` in the code above, which allows the next function to safely grab the token as an argument. To allow the possibly-`\outer` token #1 as an argument of the `\user's function` (which is protected and stored in `\l__tl_peek_code_tl`), we set it equal to a harmless macro. This must be done at the very last minute because #1 may be some pretty important function such as `\exp_after:wN`. Using a primitive `\cs_set_nopar:Npe` expansion (to avoid `\outer` problems) we set up to run the code `\let #1 \user's function \user's function` followed by arguments involving #1. Regardless of #1 (including the user's function itself), the user's function is run. It always starts with `\group_end:`, which has not been redefined since #1 started out as expandable, and which restores the definition of #1.

Then we put the elaborate first argument `\_kernel_exp_not:w \exp_after:wN { \exp_not:N #1 }`: indeed we cannot use `\exp_not:n {#1}` as this breaks for an `\outer` macro and we cannot use `\exp_not:N #1`, as o-expanding this yields a “notexpanded” token equal to (a weird) `\relax`, which would have the wrong value for primitive T_EX conditionals such as `\if_meaning:w`.

Then we must add `{-1}0` if the token is a control sequence and `{\charcode}`D otherwise. Distinguishing the two cases is easy: since we have made the escape character printable, `\token_to_str:N` gives at least two characters for a control sequence versus a single one for an active character (possibly being a space, in which case the trailing brace group is taken as the first argument of `\_tl_peek_analysis_exp_aux:Nw`). Importantly, #1 could be an `\outer` token (as it is only set to `\scan_stop:` at the last minute) but once we apply `\token_to_str:N` we no longer need to worry about it.

```

4152 \cs_new_protected:Npn \_tl_peek_analysis_exp:N #1
4153 {
4154   \cs_set_nopar:Npe \l__tl_peek_code_tl
4155   {
4156     \tex_let:D \exp_not:N #1 \l__tl_peek_code_tl
4157     \l__tl_peek_code_tl
4158     {
4159       \exp_not:n { \_kernel_exp_not:w \exp_after:wN }
4160       { \exp_not:N \exp_not:N \exp_not:N #1 }
4161     }
4162     \exp_after:wN \_tl_peek_analysis_exp_aux:Nw
4163     \token_to_str:N #1 { } \s__tl
4164   }
4165   \l__tl_peek_code_tl
4166 }
4167 \cs_new:Npe \_tl_peek_analysis_exp_aux:Nw #1#2 \s__tl
4168 {

```

```

4169 \exp_not:N \if_meaning:w \scan_stop: #2 \scan_stop:
4170 { \exp_not:N \int_value:w '#1 ~ } \token_to_str:N D
4171 \exp_not:N \else:
4172 { -1 } 0
4173 \exp_not:N \fi:
4174 }

```

For normal non-expandable tokens we must distinguish characters (including active ones and macro parameter characters) from control sequences (whose string representation is more than one character because we made the escape character printable). For a control sequence call the user code with suitable arguments, wrapping #1 within `\exp_not:n` just in case it happens to be equal to a macro parameter character. We do not skip `\exp_not:n` when unnecessary, because this auxiliary is also called in `\__tl_peek_analysis_retest:` where we have changed some control sequences or active characters to `\scan_stop:` temporarily.

```

4175 \cs_new_protected:Npn \__tl_peek_analysis_nonexp:N #1
4176 {
4177   \if_charcode:w
4178     \scan_stop:
4179     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
4180     \scan_stop:
4181     \exp_after:wN \__tl_peek_analysis_char:N
4182   \else:
4183     \exp_after:wN \__tl_peek_analysis_cs:N
4184   \fi:
4185   #1
4186 }
4187 \cs_new_protected:Npn \__tl_peek_analysis_cs:N #1
4188 { \l__tl_peek_code_tl { \exp_not:n {#1} } { -1 } 0 }

```

For normal characters we must determine their catcode. The main difficulty is that the character may be an active character masquerading as (i.e., set equal to) itself with a different catcode. Two approaches based on `\lowercase` can detect this. One could make an active character with the same catcode as #1 and change its definition before testing the catcode of #1, but in some Unicode engine this fills up the hash table uselessly. Instead, we lowercase #1 itself, changing its character code to 32, namely space (because LuaTeX cannot turn catcode 10 characters to anything else than character code 32), then we apply `\__tl_analysis_b_char:Nn`, which detects active characters by comparing them to `\tex_undefined:D`, and we must have undefined the active space (locally) for this test to work. To define `\__tl_peek_analysis_char:N` itself we use an e-expanding assignment to get the active space in the right place after making it (just for this definition) unexpandable. Finally `\__tl_peek_analysis_char:w` receives the `<charcode>`, `<user function>`, `<catcode>`, and `<token>`, and places the arguments in the correct order. It keeps `\exp_not:n` for macro parameter characters and active characters (the latter could be macro parameter characters, and it seems more uniform to always put `\exp_not:n`), and otherwise eliminates it by expanding once with `\exp_args:NNNo`.

```

4189 \group_begin:
4190 \char_set_active_eq:NN \ \scan_stop:
4191 \cs_new_protected:Npe \__tl_peek_analysis_char:N #1
4192 {
4193   \cs_set_eq:NN
4194   \char_generate:nn { 32 } { 13 }
4195   \exp_not:N \tex_undefined:D

```

```

4196 \tex_lccode:D '#1 = 32 \exp_stop_f:
4197 \tex_lowercase:D
4198 {
4199   \tl_put_right:Ne \exp_not:N \l__tl_peek_code_tl
4200   { \exp_not:n { \__tl_analysis_b_char:Nn \use_none:n } {#1} }
4201 }
4202 \exp_not:n
4203 {
4204   \exp_after:wN \__tl_peek_analysis_char:w
4205   \int_value:w
4206 }
4207 '#1
4208 \exp_not:n { \exp_after:wN \s__tl \l__tl_peek_code_tl }
4209 #1
4210 }
4211 \group_end:
4212 \cs_new_protected:Npn \__tl_peek_analysis_char:w #1 \s__tl #2#3#4
4213 {
4214   \if_charcode:w 6 #3
4215   \else:
4216     \if_charcode:w D #3
4217     \else:
4218       \exp_args:NNNo
4219       \fi:
4220     \fi:
4221     #2 { \exp_not:n {#4} } {#1} #3
4222 }

```

For special characters the idea is to eventually act with `\token_to_str:N`, then pick up one by one the characters of this string representation until hitting the token that follows. First determine the character code of (the meaning of) the `<token>` (which we know is a special token), make sure the escape character is different from it, normalize the meanings of two active characters and the empty control sequence, and filter out these cases in `\__tl_peek_analysis_retest:`.

```

4223 \cs_new_protected:Npn \__tl_peek_analysis_special:
4224 {
4225   \tex_let:D \l__tl_analysis_token = ~ \l_peek_token
4226   \int_set:Nn \l__tl_peek_charcode_int
4227   { \__tl_analysis_extract_charcode: }
4228   \if_int_compare:w \l__tl_peek_charcode_int = \tex_escapechar:D
4229   \int_set:Nn \tex_escapechar:D { '\ / }
4230   \fi:
4231   \char_set_active_eq:nN { \l__tl_peek_charcode_int } \scan_stop:
4232   \char_set_active_eq:nN { \tex_escapechar:D } \scan_stop:
4233   \cs_set_eq:cN { } \scan_stop:
4234   \tex_futurelet:D \l__tl_analysis_token
4235   \__tl_peek_analysis_retest:
4236 }
4237 \cs_new_protected:Npn \__tl_peek_analysis_retest:
4238 {
4239   \if_meaning:w \l__tl_analysis_token \scan_stop:
4240     \exp_after:wN \__tl_peek_analysis_nonexp:N
4241   \else:
4242     \exp_after:wN \__tl_peek_analysis_str:

```

```

4243     \fi:
4244 }

```

At this point we know the meaning of the $\langle token \rangle$ in the input stream is $\backslash l\_peek\_token$, either a space (32, 10) or a begin-group or end-group token (catcode 1 or 2), and we excluded a few cases that would be difficult later (empty control sequence, active character with the same character code as its meaning or as the escape character). The idea is to apply $\backslash token\_to\_str:N$ to the $\langle token \rangle$ then grab characters (of category code 12 except for spaces that have category code 10) to reconstruct it. In earlier versions of the code we would peek at the $\langle next\ token \rangle$ that lies after $\langle token \rangle$ in the input stream, which would help us be more accurate in reconstructing the $\langle token \rangle$ case in edge cases (mentioned below), but this had the side-effect of tokenizing the input stream (turning characters into tokens) farther ahead than needed.

We hit the $\langle token \rangle$ with $\backslash token\_to\_str:N$ and start grabbing characters. More precisely, by looking at the first character in the string representation of the $\langle token \rangle$ we distinguish three cases: a stringified control sequence starts with the escape character; for an explicit character we find that same character; for an active character we find anything else (we made sure to exclude the case of an active character whose string representation coincides with the other two cases).

```

4245 \cs_new_protected:Npn \__tl_peek_analysis_str:
4246 {
4247     \exp_after:wN \tex_futurelet:D
4248     \exp_after:wN \l__tl_analysis_token
4249     \exp_after:wN \__tl_peek_analysis_str:w
4250     \token_to_str:N
4251 }
4252 \cs_new_protected:Npn \__tl_peek_analysis_str:w
4253 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_str:n }
4254 \cs_new_protected:Npn \__tl_peek_analysis_str:n #1
4255 {
4256     \int_case:nnF { '#1 }
4257     {
4258         { \l__tl_peek_charcode_int }
4259         { \__tl_peek_analysis_explicit:n {#1} }
4260         { \tex_escapechar:D } { \__tl_peek_analysis_escape: }
4261     }
4262     { \__tl_peek_analysis_active_str:n {#1} }
4263 }

```

When $\#1$ is a stringified active character we pass appropriate arguments to the user's code; thankfully $\backslash char\_generate:nn$ can make active characters.

```

4264 \cs_new_protected:Npn \__tl_peek_analysis_active_str:n #1
4265 {
4266     \tl_put_right:Ne \l__tl_peek_code_tl
4267     {
4268         { \char_generate:nn { '#1 } { 13 } }
4269         { \int_value:w '#1 }
4270         \token_to_str:N D
4271     }
4272     \l__tl_peek_code_tl
4273 }

```

When $\#1$ matches the character we had extracted from the meaning of $\backslash l\_peek\_token$, the token was an explicit character, which can be a standard space, or a begin-group or

end-group character with some character code. In the latter two cases we call `\char_generate:nn` with suitable arguments and put suitable `\if_false: \fi:` constructions to make the result balanced and such that o-expanding or e/x-expanding gives back a single (unbalanced) begin-group or end-group character.

```

4274 \cs_new_protected:Npn \__tl_peek_analysis_explicit:n #1
4275 {
4276   \tl_put_right:Ne \l__tl_peek_code_tl
4277   {
4278     \if_meaning:w \l_peek_token \c_space_token
4279     { ~ } { 32 } \token_to_str:N A
4280   \else:
4281     \if_catcode:w \l_peek_token \c_group_begin_token
4282     {
4283       \exp_not:N \exp_after:wN
4284       \char_generate:nn { '#1 } { 1 }
4285       \exp_not:N \if_false:
4286       \if_false: { \fi: }
4287       \exp_not:N \fi:
4288     }
4289     { \int_value:w '#1 }
4290     1
4291   \else:
4292     {
4293       \exp_not:N \if_false:
4294       { \if_false: } \fi:
4295       \exp_not:N \fi:
4296       \char_generate:nn { '#1 } { 2 }
4297     }
4298     { \int_value:w '#1 }
4299     2
4300   \fi:
4301   \fi:
4302 }
4303 \l__tl_peek_code_tl
4304 }
```

Finally there is the case of a special token whose string representation starts with an escape character, namely the token was a control sequence. In that case we could have grabbed the token directly as an N-type argument, but of course we couldn't know that until we had run all the various tests including stringifying the token. We are thus left with the hard work of picking up one by one the characters in the csname (being careful about spaces), until the constructed csname has the expected meaning. This fails if someone defines a token like `\bgroup@my` whose string representation starts the same as another token with the same meaning being an implicit character token of category code 1, 2, or 10.

```

4305 \cs_new_protected:Npn \__tl_peek_analysis_escape:
4306 {
4307   \tl_clear:N \l__tl_tmpa_tl
4308   \tex_futurelet:D \l__tl_analysis_token
4309   \__tl_peek_analysis_collect:w
4310 }
4311 \cs_new_protected:Npn \__tl_peek_analysis_collect:w
4312 { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_collect:n }
```

```

4313 \cs_new_protected:Npn \__tl_peek_analysis_collect:n #1
4314 {
4315   \tl_put_right:Nn \l__tl_tmpa_tl {#1}
4316   \__tl_peek_analysis_collect_loop:
4317 }
4318 \cs_new_protected:Npn \__tl_peek_analysis_collect_loop:
4319 {
4320   \exp_after:wN \if_meaning:w
4321   \cs:w
4322   \if_cs_exist:w \l__tl_tmpa_tl \cs_end:
4323   \l__tl_tmpa_tl
4324   \else:
4325     c_one % anything short
4326   \fi:
4327   \cs_end:
4328   \l_peek_token
4329   \__tl_peek_analysis_collect_end:NNNN
4330 \fi:
4331 \tex_futurelet:D \l__tl_analysis_token
4332   \__tl_peek_analysis_collect:w
4333 }

```

As in all other cases, end by calling the user code with suitable arguments (here #1 is \fi:).

```

4334 \cs_new_protected:Npn \__tl_peek_analysis_collect_end:NNNN #1#2#3#4
4335 {
4336   #1
4337   \tl_put_right:Ne \l__tl_peek_code_tl
4338   {
4339     { \exp_not:N \exp_not:n { \exp_not:c { \l__tl_tmpa_tl } } }
4340     { -1 }
4341     0
4342   }
4343   \l__tl_peek_code_tl
4344 }

```

(End of definition for \peek_analysis_map_inline:n and others. This function is documented on page 214.)

49.11 Messages

\c__tl_analysis_show_etc_str When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

4345 \tl_const:Ne \c__tl_analysis_show_etc_str % (
4346   { \token_to_str:N \ETC.) }

```

(End of definition for \c__tl_analysis_show_etc_str.)

```

4347 \msg_new:nnn { tl } { show-analysis }
4348 {
4349   The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
4350   \tl_if_empty:nTF {#2}
4351     { is~empty #3 }
4352     { contains~the~tokens: #2 #4 }
4353 }

```


4354 `</code>`

Chapter 50

l3regex implementation

```
4355 <*code>
4356 <@@=regex>
```

50.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since T_EX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyze the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labeled by an integer $\langle position \rangle$, with $\min\_pos - 1 \leq \langle position \rangle \leq \max\_pos$. The lowest and highest positions $\min\_pos - 1$ and $\max\_pos$ correspond to imaginary begin and end markers (with non-existent category code and character code). $\max\_pos$ is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labeled by an integer $\langle state \rangle$ with $\min\_state \leq \langle state \rangle < \max\_state$.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.

- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use `\intarray` to manipulate arrays of integers. We also abuse $\text{\TeX}$'s `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks<state>` holds the tests and actions to perform in the `<state>` of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last `<step>` in which each `<state>` was active.
- `\g__regex_thread_info_intarray` consists of blocks for each `<thread>` (with $\text{min\_thread} \leq \text{<thread>} < \text{max\_thread}$). Each block has $1+2\text{\l__regex\_capturing\_group\_int}$ entries: the `<state>` in which the `<thread>` currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The `<threads>` are ordered starting from the best to the least preferred.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks<position>` holds `<tokens>` which o- and e-expand to the `<position>`-th token in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

50.2 Helpers

`\__regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

4357 `\cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D`

(End of definition for `\__regex_int_eval:w`.)

`\__regex_sep:`

4358 `\cs_new_eq:NN \__regex_sep: \__kernel_int_sep:`

(End of definition for `\_regex_sep:`.)

`\_regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.

```
4359 \cs_new_protected:Npn \_regex_standard_escapechar:
4360 { \int_set:Nn \tex_escapechar:D { '\ } }
```

(End of definition for `\_regex_standard_escapechar:`.)

`\_regex_toks_use:w` Unpack a `\toks` given its number.

```
4361 \cs_new:Npn \_regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(End of definition for `\_regex_toks_use:w`.)

`\_regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.

```
\_regex_toks_set:Nn
\_regex_toks_set:No
4362 \cs_new_protected:Npn \_regex_toks_clear:N #1
4363 { \tex_toks:D #1 = { } }
4364 \cs_new_eq:NN \_regex_toks_set:Nn \tex_toks:D
4365 \cs_new_protected:Npn \_regex_toks_set:No #1
4366 { \tex_toks:D #1 = \exp_after:wN }
```

(End of definition for `\_regex_toks_clear:N` and `\_regex_toks_set:Nn`.)

`\_regex_toks_memcpy:NNn` Copy #3 `\toks` registers from #2 onwards to #1 onwards, like C's `memcpy`.

```
4367 \cs_new_protected:Npn \_regex_toks_memcpy:NNn #1#2#3
4368 {
4369   \prg_replicate:nn {#3}
4370   {
4371     \tex_toks:D #1 = \tex_toks:D #2
4372     \int_incr:N #1
4373     \int_incr:N #2
4374   }
4375 }
```

(End of definition for `\_regex_toks_memcpy:NNn`.)

`\_regex_toks_put_left:Ne` During the building phase we wish to add e-expanded material to `\toks`, either to the left or to the right. The expansion is done “by hand” for optimization (these operations are used quite a lot). The `Nn` version of `\_regex_toks_put_right:Ne` is provided because it is more efficient than e-expanding with `\exp_not:n`.

```
4376 \cs_if_exist:NTF \tex_etokspre:D
4377 { \cs_new_eq:NN \_regex_toks_put_left:Ne \tex_etokspre:D }
4378 {
4379   \cs_new_protected:Npn \_regex_toks_put_left:Ne #1#2
4380   { \tex_toks:D #1 = \tex_expanded:D {{ #2 \tex_the:D \tex_toks:D #1 }} }
4381 }
4382 \cs_if_exist:NTF \tex_etoksapp:D
4383 { \cs_new_eq:NN \_regex_toks_put_right:Ne \tex_etoksapp:D }
4384 {
4385   \cs_new_protected:Npn \_regex_toks_put_right:Ne #1#2
4386   { \tex_toks:D #1 = \tex_expanded:D {{ \tex_the:D \tex_toks:D #1 #2 }} }
4387 }
4388 \cs_if_exist:NTF \tex_toksapp:D
4389 { \cs_new_eq:NN \_regex_toks_put_right:Nn \tex_toksapp:D }
4390 {
4391   \cs_new_protected:Npn \_regex_toks_put_right:Nn #1#2
```

```

4392     { \tex_toks:D #1 = \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
4393   }

```

(End of definition for _regex_toks_put_left:Ne and _regex_toks_put_right:Ne.)

_regex_curr_cs_to_str: Expands to the string representation of the token (known to be a control sequence) at the current position `\l\_regex_curr_pos_int`. It should only be used in `e/x`-expansion to avoid losing a leading space.

```

4394 \cs_new:Npn \_regex_curr_cs_to_str:
4395 {
4396   \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
4397   \l\_regex_curr_token_tl
4398 }

```

(End of definition for _regex_curr_cs_to_str:.)

_regex_intarray_item:NnF Item of intarray, with a default value.

```

\_regex_intarray_item_aux:nNF
4399 \cs_new:Npn \_regex_intarray_item:NnF #1#2
4400 { \exp_args:No \_regex_intarray_item_aux:nNF { \tex_the:D \_regex_int_eval:w #2 } #1 }
4401 \cs_new:Npn \_regex_intarray_item_aux:nNF #1#2
4402 {
4403   \if_int_compare:w #1 > \c_zero_int
4404     \exp_after:wN \use_ii:nnn
4405   \fi:
4406   \use_ii:nn { \_kernel_intarray_item:Nn #2 {#1} }
4407 }

```

(End of definition for _regex_intarray_item:NnF and _regex_intarray_item_aux:nNF.)

_regex_maplike_break: Analogous to `\tl_map_break:`, this correctly exits `\tl_map_inline:nn` and similar constructions and jumps to the matching `\prg_break_point:Nn \_regex_maplike_break: { }`.

```

4408 \cs_new:Npn \_regex_maplike_break:
4409 { \prg_map_break:Nn \_regex_maplike_break: { } }

```

(End of definition for _regex_maplike_break:.)

_regex_tl_odd_items:n **_regex_tl_even_items:n** Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```

\_regex_tl_even_items_loop:nn
4410 \cs_new:Npn \_regex_tl_odd_items:n #1 { \_regex_tl_even_items:n { ? #1 } }
4411 \cs_new:Npn \_regex_tl_even_items:n #1
4412 {
4413   \_regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
4414   \prg_break_point:
4415 }
4416 \cs_new:Npn \_regex_tl_even_items_loop:nn #1#2
4417 {
4418   \_regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
4419   { \exp_not:n {#2} }
4420   \_regex_tl_even_items_loop:nn
4421 }

```

(End of definition for _regex_tl_odd_items:n, _regex_tl_even_items:n, and _regex_tl_even_items_loop:nn.)

50.2.1 Constants and variables

`\__regex_tmp:w` Temporary function used for various short-term purposes.

```
4422 \cs_new:Npn \__regex_tmp:w { }
```

(End of definition for __regex_tmp:w.)

`\l__regex_tmpa_tl` Temporary variables used for various purposes.

```
\l__regex_tmpb_tl 4423 \tl_new:N \l__regex_tmpa_tl
\l__regex_tmpa_int 4424 \tl_new:N \l__regex_tmpb_tl
\l__regex_tmpb_int 4425 \int_new:N \l__regex_tmpa_int
\l__regex_tmppc_int 4426 \int_new:N \l__regex_tmpb_int
\l__regex_tmp_bool 4427 \int_new:N \l__regex_tmppc_int
\l__regex_tmp_seq 4428 \bool_new:N \l__regex_tmp_bool
\g__regex_tmp_tl 4429 \seq_new:N \l__regex_tmp_seq
4430 \tl_new:N \g__regex_tmp_tl
```

(End of definition for \l__regex_tmpa_tl and others.)

`\l__regex_build_tl` This temporary variable is specifically for use with the `tl_build` machinery.

```
4431 \tl_new:N \l__regex_build_tl
```

(End of definition for \l__regex_build_tl.)

`\c__regex_no_match_regex` This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
4432 \tl_const:Nn \c__regex_no_match_regex
4433 {
4434   \__regex_branch:n
4435   { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
4436 }
```

(End of definition for \c__regex_no_match_regex.)

`\l__regex_balance_int` During this phase, `\l__regex_balance_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
4437 \int_new:N \l__regex_balance_int
```

(End of definition for \l__regex_balance_int.)

50.2.2 Testing characters

```
\c__regex_ascii_min_int
\c__regex_ascii_max_control_int 4438 \int_const:Nn \c__regex_ascii_min_int { 0 }
\c__regex_ascii_max_int 4439 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
4440 \int_const:Nn \c__regex_ascii_max_int { 127 }
```

(End of definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex_ascii_max_int.)

```
\c__regex_ascii_lower_int
4441 \int_const:Nn \c__regex_ascii_lower_int { 'a - 'A }
```

(End of definition for \c__regex_ascii_lower_int.)

50.2.3 Internal auxiliaries

`\q__regex_recursion_stop` Internal recursion quarks.

```
4442 \quark_new:N \q__regex_recursion_stop
```

(End of definition for `\q__regex_recursion_stop`.)

`\q__regex_nil` Internal quarks.

```
4443 \quark_new:N \q__regex_nil
```

(End of definition for `\q__regex_nil`.)

`\__regex_use_none_delimit_by_q_recursion_stop:w` Functions to gobble up to a quark.

`\__regex_use_i_delimit_by_q_recursion_stop:nw`

`\__regex_use_none_delimit_by_q_nil:w`

```
4444 \cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
```

```
4445   #1 \q__regex_recursion_stop { }
```

```
4446 \cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
```

```
4447   #1 #2 \q__regex_recursion_stop {#1}
```

```
4448 \cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }
```

(End of definition for `\__regex_use_none_delimit_by_q_recursion_stop:w`, `\__regex_use_i_delimit_by_q_recursion_stop:nw`, and `\__regex_use_none_delimit_by_q_nil:w`.)

`\__regex_quark_if_nil_p:n` Branching quark conditional.

`\__regex_quark_if_nil:nTF`

```
4449 \__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
```

(End of definition for `\__regex_quark_if_nil:nTF`.)

`\__regex_break_point:TF`

`\__regex_break_true:w`

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

```
<test1> ... <test_n>
__regex_break_point:TF {<true code>} {<false code>}
```

If any of the tests succeeds, it calls `\__regex_break_true:w`, which cleans up and leaves `<true code>` in the input stream. Otherwise, `\__regex_break_point:TF` leaves the `<false code>` in the input stream.

```
4450 \cs_new_protected:Npn \__regex_break_true:w
```

```
4451   #1 \__regex_break_point:TF #2 #3 {#2}
```

```
4452 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(End of definition for `\__regex_break_point:TF` and `\__regex_break_true:w`.)

`\__regex_item_reverse:n`

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by `-2`, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
4453 \cs_new_protected:Npn \__regex_item_reverse:n #1
```

```
4454   {
```

```
4455     #1
```

```
4456     \__regex_break_point:TF { } \__regex_break_true:w
```

```
4457   }
```

(End of definition for `\__regex_item_reverse:n`.)

_regex_item_caseful_equal:n
_regex_item_caseful_range:nn

Simple comparisons triggering _regex_break_true:w when true.

```

4458 \cs_new_protected:Npn \_regex_item_caseful_equal:n #1
4459 {
4460   \if_int_compare:w #1 = \l__regex_curr_char_int
4461     \exp_after:wN \_regex_break_true:w
4462   \fi:
4463 }
4464 \cs_new_protected:Npn \_regex_item_caseful_range:nn #1 #2
4465 {
4466   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4467   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4468   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4469   \fi:
4470   \fi:
4471 }

```

(End of definition for _regex_item_caseful_equal:n and _regex_item_caseful_range:nn.)

_regex_item_caseless_equal:n
_regex_item_caseless_range:nn

For caseless matching, we perform the test both on the curr_char and on the case_changed_char. Before doing the second set of tests, we make sure that case_changed_char has been computed.

```

4472 \cs_new_protected:Npn \_regex_item_caseless_equal:n #1
4473 {
4474   \if_int_compare:w #1 = \l__regex_curr_char_int
4475     \exp_after:wN \_regex_break_true:w
4476   \fi:
4477   \_regex_maybe_compute_ccc:
4478   \if_int_compare:w #1 = \l__regex_case_changed_char_int
4479     \exp_after:wN \_regex_break_true:w
4480   \fi:
4481 }
4482 \cs_new_protected:Npn \_regex_item_caseless_range:nn #1 #2
4483 {
4484   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4485   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4486   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4487   \fi:
4488   \fi:
4489   \_regex_maybe_compute_ccc:
4490   \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
4491   \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
4492   \exp_after:wN \exp_after:wN \exp_after:wN \_regex_break_true:w
4493   \fi:
4494   \fi:
4495 }

```

(End of definition for _regex_item_caseless_equal:n and _regex_item_caseless_range:nn.)

_regex_compute_case_changed_char:

This function is called when \l__regex_case_changed_char_int has not yet been computed. If the current character code is in the range [65, 90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97, 122], subtract 32.

```

4496 \cs_new_protected:Npn \_regex_compute_case_changed_char:
4497 {
4498   \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int

```



```

4499 \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
4500 \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
4501 \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
4502 \int_sub:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
4503 \fi:
4504 \fi:
4505 \else:
4506 \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
4507 \int_add:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
4508 \fi:
4509 \fi:
4510 \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
4511 }
4512 \cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:

```

(End of definition for `\__regex_compute_case_changed_char:.`)

`\__regex_item_equal:n` Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

4513 \cs_new_eq:NN \__regex_item_equal:n ?
4514 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End of definition for `\__regex_item_equal:n` and `\__regex_item_range:nn`.)

`\__regex_item_catcode:nT` The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

4515 \cs_new_protected:Npn \__regex_item_catcode:
4516 {
4517 "
4518 \if_case:w \l__regex_curr_catcode_int
4519 1 \or: 4 \or: 10 \or: 40
4520 \or: 100 \or: 1000 \or: 4000
4521 \or: 10000 \or: 100000 \or: 400000
4522 \or: 1000000 \or: 4000000 \else: 1*0
4523 \fi:
4524 }
4525 \prg_new_protected_conditional:Npnn \__regex_item_catcode:n #1 { T }
4526 {
4527 \if_int_odd:w \__regex_int_eval:w #1 / \__regex_item_catcode: \scan_stop:
4528 \prg_return_true:
4529 \else:
4530 \prg_return_false:
4531 \fi:
4532 }
4533 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
4534 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(End of definition for `\__regex_item_catcode:nT`, `\__regex_item_catcode_reverse:nT`, and `\__regex_item_catcode:.`)

`\\_regex_item_exact:nn` This matches an exact $\langle category \rangle$ - $\langle character\ code \rangle$ pair, or an exact control sequence, more precisely one of several possible control sequences, separated by `\\scan_stop:`.

```

4535 \\cs_new_protected:Npn \\_regex_item_exact:nn #1#2
4536 {
4537   \\if_int_compare:w #1 = \\l__regex_curr_catcode_int
4538   \\if_int_compare:w #2 = \\l__regex_curr_char_int
4539   \\exp_after:wN \\exp_after:wN \\exp_after:wN \\_regex_break_true:w
4540   \\fi:
4541   \\fi:
4542 }
4543 \\cs_new_protected:Npn \\_regex_item_exact_cs:n #1
4544 {
4545   \\int_compare:nNnTF \\l__regex_curr_catcode_int = \\c_zero_int
4546   {
4547     \\_kernel_tl_set:Nx \\l__regex_tmpa_tl
4548     { \\scan_stop: \\_regex_curr_cs_to_str: \\scan_stop: }
4549     \\tl_if_in:noTF { \\scan_stop: #1 \\scan_stop: }
4550     \\l__regex_tmpa_tl
4551     { \\_regex_break_true:w } { }
4552   }
4553   { }
4554 }
```

(End of definition for `\\_regex_item_exact:nn` and `\\_regex_item_exact_cs:n`.)

`\\_regex_item_cs:n` Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```

4555 \\cs_new_protected:Npn \\_regex_item_cs:n #1
4556 {
4557   \\int_compare:nNnT \\l__regex_curr_catcode_int = \\c_zero_int
4558   {
4559     \\group_begin:
4560     \\_regex_single_match:
4561     \\_regex_disable_submatches:
4562     \\_regex_build_for_cs:n {#1}
4563     \\bool_set_eq:NN \\l__regex_saved_success_bool
4564     \\g__regex_success_bool
4565     \\exp_args:Ne \\_regex_match_cs:n { \\_regex_curr_cs_to_str: }
4566     \\if_meaning:w \\c_true_bool \\g__regex_success_bool
4567     \\group_insert_after:N \\_regex_break_true:w
4568     \\fi:
4569     \\bool_gset_eq:NN \\g__regex_success_bool
4570     \\l__regex_saved_success_bool
4571     \\group_end:
4572   }
4573 }
```

(End of definition for `\\_regex_item_cs:n`.)

50.2.4 Character property tests

`\\_regex_prop_d:` Character property tests for `\\d`, `\\W`, etc. These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\\d=[0-9]`,
`\\_regex_prop_h:`
`\\_regex_prop_s:`
`\\_regex_prop_v:`
`\\_regex_prop_w:`
`\\_regex_prop_N:`

`\w=[0-9A-Z_a-z]`, `\s=[\_\^\^I\^J\^L\^M]`, `\h=[\_\^\^I]`, `\v=[\^J-\^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

4574 \cs_new_protected:Npn \__regex_prop_d:
4575 { \__regex_item_caseful_range:nn { '0 } { '9 } }
4576 \cs_new_protected:Npn \__regex_prop_h:
4577 {
4578   \__regex_item_caseful_equal:n { '\ }
4579   \__regex_item_caseful_equal:n { '\^I }
4580 }
4581 \cs_new_protected:Npn \__regex_prop_s:
4582 {
4583   \__regex_item_caseful_equal:n { '\ }
4584   \__regex_item_caseful_equal:n { '\^I }
4585   \__regex_item_caseful_equal:n { '\^J }
4586   \__regex_item_caseful_equal:n { '\^L }
4587   \__regex_item_caseful_equal:n { '\^M }
4588 }
4589 \cs_new_protected:Npn \__regex_prop_v:
4590 { \__regex_item_caseful_range:nn { '\^J } { '\^M } } % lf, vtab, ff, cr
4591 \cs_new_protected:Npn \__regex_prop_w:
4592 {
4593   \__regex_item_caseful_range:nn { 'a } { 'z }
4594   \__regex_item_caseful_range:nn { 'A } { 'Z }
4595   \__regex_item_caseful_range:nn { '0 } { '9 }
4596   \__regex_item_caseful_equal:n { '_' }
4597 }
4598 \cs_new_protected:Npn \__regex_prop_N:
4599 {
4600   \__regex_item_reverse:n
4601   { \__regex_item_caseful_equal:n { '\^J } }
4602 }

```

(End of definition for `\__regex_prop_d:` and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha: 4603 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 4604 { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 4605 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 4606 { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 4607 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 4608 {
\__regex_posix_lower: 4609   \__regex_item_caseful_range:nn
\__regex_posix_print: 4610   \c__regex_ascii_min_int
\__regex_posix_punct: 4611   \c__regex_ascii_max_int
\__regex_posix_space: 4612 }
\__regex_posix_upper: 4613 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_word: 4614 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_xdigit: 4615 {
4616   \__regex_item_caseful_range:nn
4617   \c__regex_ascii_min_int
4618   \c__regex_ascii_max_control_int
4619   \__regex_item_caseful_equal:n \c__regex_ascii_max_int
4620 }

```

```

4621 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
4622 \cs_new_protected:Npn \__regex_posix_graph:
4623   { \__regex_item_caseful_range:nn { '!' } { '~ } }
4624 \cs_new_protected:Npn \__regex_posix_lower:
4625   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
4626 \cs_new_protected:Npn \__regex_posix_print:
4627   { \__regex_item_caseful_range:nn { '\' } { '~ } }
4628 \cs_new_protected:Npn \__regex_posix_punct:
4629   {
4630     \__regex_item_caseful_range:nn { '!' } { '/' }
4631     \__regex_item_caseful_range:nn { ':' } { '@' }
4632     \__regex_item_caseful_range:nn { '[' } { '"' }
4633     \__regex_item_caseful_range:nn { '\{ } { '~ }
4634   }
4635 \cs_new_protected:Npn \__regex_posix_space:
4636   {
4637     \__regex_item_caseful_equal:n { '\' }
4638     \__regex_item_caseful_range:nn { '^I } { '^M }
4639   }
4640 \cs_new_protected:Npn \__regex_posix_upper:
4641   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
4642 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
4643 \cs_new_protected:Npn \__regex_posix_xdigit:
4644   {
4645     \__regex_posix_digit:
4646     \__regex_item_caseful_range:nn { 'A' } { 'F' }
4647     \__regex_item_caseful_range:nn { 'a' } { 'f' }
4648   }

```

(End of definition for `\__regex_posix_alnum:` and others.)

50.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, etc. In this pass, we also convert any special character (`*`, `?`, `{`, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\__regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *{<token list>}* The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<inline 1>*, and escaped characters are fed to the function *<inline 2>* within an *e*-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<inline 3>*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an *e*-expanding assignment.

`\__regex_escape_use:nnnn` The result is built in `\l__regex_tmpa_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through #4 once, applying #1, #2,

or #3 as relevant to each character (after de-escaping it).

```

4649 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
4650 {
4651   \group_begin:
4652     \tl_clear:N \l__regex_tmpa_tl
4653     \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
4654     \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
4655     \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
4656     \__regex_standard_escapechar:
4657     \__kernel_tl_gset:Nx \g__regex_tmp_tl
4658       { \__kernel_str_to_other_fast:n {#4} }
4659     \tl_put_right:Ne \l__regex_tmpa_tl
4660     {
4661       \exp_after:wN \__regex_escape_loop:N \g__regex_tmp_tl
4662       \scan_stop: \prg_break_point:
4663     }
4664     \exp_after:wN
4665     \group_end:
4666     \l__regex_tmpa_tl
4667   }

```

(End of definition for __regex_escape_use:nnnn.)

__regex_escape_loop:N __regex_escape_loop:N reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

4668 \cs_new:Npn \__regex_escape_loop:N #1
4669 {
4670   \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
4671     { \__regex_escape_unescaped:N #1 }
4672   \__regex_escape_loop:N
4673 }
4674 \cs_new:cpn { __regex_escape_ \c_backslash_str :w }
4675   \__regex_escape_loop:N #1
4676 {
4677   \cs_if_exist_use:cF { __regex_escape_/token_to_str:N #1:w }
4678     { \__regex_escape_escaped:N #1 }
4679   \__regex_escape_loop:N
4680 }

```

(End of definition for __regex_escape_loop:N and __regex_escape_\:w.)

__regex_escape_unescaped:N Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

\__regex_escape_escaped:N
\__regex_escape_raw:N
4681 \cs_new_eq:NN \__regex_escape_unescaped:N ?
4682 \cs_new_eq:NN \__regex_escape_escaped:N ?
4683 \cs_new_eq:NN \__regex_escape_raw:N ?

```

(End of definition for __regex_escape_unescaped:N, __regex_escape_escaped:N, and __regex_escape_raw:N.)

__regex_escape_\scan_stop:w The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

\__regex_escape_/scan_stop:w
\__regex_escape_/a:w
\__regex_escape_/e:w
\__regex_escape_/f:w
\__regex_escape_/n:w
\__regex_escape_/r:w
\__regex_escape_/t:w
\__regex_escape_\:w
4684 \cs_new_eq:cN { __regex_escape_ \iow_char:N\scan_stop: :w } \prg_break:

```

```

4685 \cs_new:cpn { __regex_escape_/ \iow_char:N\scan_stop: :w }
4686 {
4687     \msg_expandable_error:nn { regex } { trailing-backslash }
4688     \prg_break:
4689 }
4690 \cs_new:cpn { __regex_escape_~:w } { }
4691 \cs_new:cpe { __regex_escape_/a:w }
4692 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^G }
4693 \cs_new:cpe { __regex_escape_/t:w }
4694 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^I }
4695 \cs_new:cpe { __regex_escape_/n:w }
4696 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^J }
4697 \cs_new:cpe { __regex_escape_/f:w }
4698 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^L }
4699 \cs_new:cpe { __regex_escape_/r:w }
4700 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^M }
4701 \cs_new:cpe { __regex_escape_/e:w }
4702 { \exp_not:N __regex_escape_raw:N \iow_char:N ^^[ }

```

(End of definition for `__regex_escape_\scan_stop::w` and others.)

```

__regex_escape_/x:w
__regex_escape_x_end:w
__regex_escape_x_large:n

```

When `\x` is encountered, `__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `__regex_escape_raw:N` on the corresponding character token.

```

4703 \cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
4704 {
4705     \exp_after:wN __regex_escape_x_end:w
4706     \int_value:w "0 __regex_escape_x_test:N
4707 }
4708 \cs_new:Npn __regex_escape_x_end:w #1 __regex_sep:
4709 {
4710     \int_compare:nNnTF {#1} > \c_max_char_int
4711     {
4712         \msg_expandable_error:nnff { regex } { x-overflow }
4713         {#1} { \int_to_Hex:n {#1} }
4714     }
4715     {
4716         \exp_last_unbraced:Nf __regex_escape_raw:N
4717         { \char_generate:nn {#1} { 12 } }
4718     }
4719 }

```

(End of definition for `__regex_escape_/x:w`, `__regex_escape_x_end:w`, and `__regex_escape_x_large:n`.)

```

__regex_escape_x_test:N
__regex_escape_x_testii:N

```

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `__regex_escape_x_loop:N` or `__regex_escape_x:N`.

```

4720 \cs_new:Npn __regex_escape_x_test:N #1
4721 {
4722     \if_meaning:w \scan_stop: #1
4723     \exp_after:wN \use_i:nnn \exp_after:wN __regex_sep:

```

```

4724 \fi:
4725 \use:n
4726 {
4727   \if_charcode:w \c_space_token #1
4728   \exp_after:wN \_\_regex_escape_x_test:N
4729   \else:
4730     \exp_after:wN \_\_regex_escape_x_testii:N
4731     \exp_after:wN #1
4732   \fi:
4733 }
4734 }
4735 \cs_new:Npn \_\_regex_escape_x_testii:N #1
4736 {
4737   \if_charcode:w \c_left_brace_str #1
4738   \exp_after:wN \_\_regex_escape_x_loop:N
4739   \else:
4740     \_\_regex_hexadecimal_use:NTF #1
4741     { \exp_after:wN \_\_regex_escape_x:N }
4742     { \_\_regex_sep: \exp_after:wN \_\_regex_escape_loop:N \exp_after:wN #1 }
4743   \fi:
4744 }

```

(End of definition for __regex_escape_x_test:N and __regex_escape_x_testii:N.)

__regex_escape_x:N This looks for the second digit in the unbraced case.

```

4745 \cs_new:Npn \_\_regex_escape_x:N #1
4746 {
4747   \if_meaning:w \scan_stop: #1
4748   \exp_after:wN \use_i:nnn \exp_after:wN \_\_regex_sep:
4749   \fi:
4750   \use:n
4751   {
4752     \_\_regex_hexadecimal_use:NTF #1
4753     { \_\_regex_sep: \_\_regex_escape_loop:N }
4754     { \_\_regex_sep: \_\_regex_escape_loop:N #1 }
4755   }
4756 }

```

(End of definition for __regex_escape_x:N.)

__regex_escape_x_loop:N Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace,
 __regex_escape_x_loop_error: otherwise raise an error outside the assignment.

```

4757 \cs_new:Npn \_\_regex_escape_x_loop:N #1
4758 {
4759   \if_meaning:w \scan_stop: #1
4760   \exp_after:wN \use_ii:nnn
4761   \fi:
4762   \use_ii:nn
4763   { \_\_regex_sep: \_\_regex_escape_x_loop_error:n { } {#1} }
4764   {
4765     \_\_regex_hexadecimal_use:NTF #1
4766     { \_\_regex_escape_x_loop:N }
4767     {
4768       \token_if_eq_charcode:NNTF \c_space_token #1

```

```

4769         { \_regex_escape_x_loop:N }
4770     {
4771         \_regex_sep:
4772         \exp_after:wN
4773         \token_if_eq_charcode:NNTF \c_right_brace_str #1
4774         { \_regex_escape_loop:N }
4775         { \_regex_escape_x_loop_error:n {#1} }
4776     }
4777 }
4778 }
4779 }
4780 \cs_new:Npn \_regex_escape_x_loop_error:n #1
4781 {
4782     \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
4783     \_regex_escape_loop:N #1
4784 }

```

(End of definition for _regex_escape_x_loop:N and _regex_escape_x_loop_error:.)

_regex_hexadecimal_use:NTF TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

4785 \cs_new:Npn \_regex_hexadecimal_use:NTF #1
4786 {
4787     \if_int_compare:w \c_one_int < "1 \token_to_str:N #1 \exp_stop_f:
4788     #1
4789     \else:
4790         \if_case:w
4791             \_regex_int_eval:w \exp_after:wN ‘ \token_to_str:N #1 - ‘a \scan_stop:
4792             A
4793             \or: B
4794             \or: C
4795             \or: D
4796             \or: E
4797             \or: F
4798             \else:
4799                 \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
4800             \fi:
4801         \fi:
4802         \use_i:nn
4803     }

```

(End of definition for _regex_hexadecimal_use:NTF.)

_regex_char_if_alphanumeric:NTF These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumeric characters are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ASCII characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ASCII are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

4804 \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
4805 {
4806   \if:w
4807     T
4808     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4809     \if_int_compare:w '#1 > 'z \exp_stop_f:
4810     \if_int_compare:w '#1 < \c__regex_ascii_max_int
4811     \else: F \fi:
4812   \else:
4813     \if_int_compare:w '#1 < 'a \exp_stop_f:
4814     \else: F \fi:
4815   \fi:
4816 \else:
4817   \if_int_compare:w '#1 > '9 \exp_stop_f:
4818   \if_int_compare:w '#1 < 'A \exp_stop_f:
4819   \else: F \fi:
4820 \else:
4821   \if_int_compare:w '#1 < '0 \exp_stop_f:
4822   \if_int_compare:w '#1 < '\' \exp_stop_f:
4823   F \fi:
4824   \else: F \fi:
4825 \fi:
4826 \fi:
4827 T
4828 \prg_return_true:
4829 \else:
4830 \prg_return_false:
4831 \fi:
4832 }
4833 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
4834 {
4835   \if:w
4836     T
4837     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4838     \if_int_compare:w '#1 > 'z \exp_stop_f:
4839     F
4840   \else:
4841     \if_int_compare:w '#1 < 'a \exp_stop_f:
4842     F \fi:
4843   \fi:
4844 \else:
4845   \if_int_compare:w '#1 > '9 \exp_stop_f:
4846   \if_int_compare:w '#1 < 'A \exp_stop_f:
4847   F \fi:
4848 \else:
4849   \if_int_compare:w '#1 < '0 \exp_stop_f:
4850   F \fi:
4851 \fi:
4852 \fi:
4853 T

```

```

4854     \prg_return_true:
4855 \else:
4856     \prg_return_false:
4857 \fi:
4858 }

```

(End of definition for `\_regex_char_if_alphanumeric:NTF` and `\_regex_char_if_special:NTF`.)

50.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\_regex_class:NnnnN` $\langle\text{boolean}\rangle$ $\{\langle\text{tests}\rangle\}$ $\{\langle\text{min}\rangle\}$ $\{\langle\text{more}\rangle\}$ $\langle\text{laziness}\rangle$
- `\_regex_group:nnnN` $\{\langle\text{branches}\rangle\}$ $\{\langle\text{min}\rangle\}$ $\{\langle\text{more}\rangle\}$ $\langle\text{laziness}\rangle$, also `\_regex_group_no_capture:nnnN` and `\_regex_group_resetting:nnnN` with the same syntax.
- `\_regex_branch:n` $\{\langle\text{contents}\rangle\}$
- `\_regex_command_K:`
- `\_regex_assertion:Nn` $\langle\text{boolean}\rangle$ $\{\langle\text{assertion test}\rangle\}$, where the $\langle\text{assertion test}\rangle$ is `\_regex_b_test:` or `\_regex_Z_test:` or `\_regex_A_test:` or `\_regex_G_test:`

Tests can be the following:

- `\_regex_item_caseful_equal:n` $\{\langle\text{char code}\rangle\}$
- `\_regex_item_caseless_equal:n` $\{\langle\text{char code}\rangle\}$
- `\_regex_item_caseful_range:nn` $\{\langle\text{min}\rangle\}$ $\{\langle\text{max}\rangle\}$
- `\_regex_item_caseless_range:nn` $\{\langle\text{min}\rangle\}$ $\{\langle\text{max}\rangle\}$
- `\_regex_item_catcode:nT` $\{\langle\text{catcode bitmap}\rangle\}$ $\{\langle\text{tests}\rangle\}$
- `\_regex_item_catcode_reverse:nT` $\{\langle\text{catcode bitmap}\rangle\}$ $\{\langle\text{tests}\rangle\}$
- `\_regex_item_reverse:n` $\{\langle\text{tests}\rangle\}$
- `\_regex_item_exact:nn` $\{\langle\text{catcode}\rangle\}$ $\{\langle\text{char code}\rangle\}$
- `\_regex_item_exact_cs:n` $\{\langle\text{csnames}\rangle\}$, more precisely given as $\langle\text{csname}\rangle$ `\scan_stop:` $\langle\text{csname}\rangle$ `\scan_stop:` $\langle\text{csname}\rangle$ and so on in a brace group.
- `\_regex_item_cs:n` $\{\langle\text{compiled regex}\rangle\}$

50.3.1 Variables used when compiling

`\l__regex_group_level_int` We make sure to open the same number of groups as we close.

```
4859 \int_new:N \l__regex_group_level_int
```

(End of definition for \l__regex_group_level_int.)

`\l__regex_mode_int`
`\c__regex_cs_in_class_mode_int`
`\c__regex_cs_mode_int`
`\c__regex_outer_mode_int`
`\c__regex_catcode_mode_int`
`\c__regex_class_mode_int`
`\c__regex_catcode_in_class_mode_int`

While compiling, ten modes are recognized, labeled -63 , -23 , -6 , -2 , 0 , 2 , 3 , 6 , 23 , 63 . See section 50.3.3. We only define some of these as constants.

```
4860 \int_new:N \l__regex_mode_int
4861 \int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
4862 \int_const:Nn \c__regex_cs_mode_int { -2 }
4863 \int_const:Nn \c__regex_outer_mode_int { 0 }
4864 \int_const:Nn \c__regex_catcode_mode_int { 2 }
4865 \int_const:Nn \c__regex_class_mode_int { 3 }
4866 \int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(End of definition for \l__regex_mode_int and others.)

`\l__regex_catcodes_int`
`\l__regex_default_catcodes_int`
`\l__regex_catcodes_bool`

We wish to allow constructions such as `\c[^BE](. . \cL[a-z] . .)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
4867 \int_new:N \l__regex_catcodes_int
4868 \int_new:N \l__regex_default_catcodes_int
4869 \bool_new:N \l__regex_catcodes_bool
```

(End of definition for \l__regex_catcodes_int, \l__regex_default_catcodes_int, and \l__regex_catcodes_bool.)

`\c__regex_catcode_C_int`
`\c__regex_catcode_B_int`
`\c__regex_catcode_E_int`
`\c__regex_catcode_M_int`
`\c__regex_catcode_T_int`
`\c__regex_catcode_P_int`
`\c__regex_catcode_U_int`
`\c__regex_catcode_D_int`
`\c__regex_catcode_S_int`
`\c__regex_catcode_L_int`
`\c__regex_catcode_O_int`
`\c__regex_catcode_A_int`
`\c__regex_all_catcodes_int`

Constants: 4^c for each category, and the sum of all powers of 4.

```
4870 \int_const:Nn \c__regex_catcode_C_int { "1 }
4871 \int_const:Nn \c__regex_catcode_B_int { "4 }
4872 \int_const:Nn \c__regex_catcode_E_int { "10 }
4873 \int_const:Nn \c__regex_catcode_M_int { "40 }
4874 \int_const:Nn \c__regex_catcode_T_int { "100 }
4875 \int_const:Nn \c__regex_catcode_P_int { "1000 }
4876 \int_const:Nn \c__regex_catcode_U_int { "4000 }
4877 \int_const:Nn \c__regex_catcode_D_int { "10000 }
4878 \int_const:Nn \c__regex_catcode_S_int { "100000 }
4879 \int_const:Nn \c__regex_catcode_L_int { "400000 }
4880 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
4881 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
4882 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(End of definition for \c__regex_catcode_C_int and others.)

`\l__regex_tmp_regex`

The compilation step stores its result in this variable.

```
4883 \cs_new_eq:NN \l__regex_tmp_regex \c__regex_no_match_regex
```

(End of definition for \l__regex_tmp_regex.)

`\l__regex_show_prefix_seq` This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
4884 \seq_new:N \l__regex_show_prefix_seq
```

(End of definition for \l__regex_show_prefix_seq.)

`\l__regex_show_lines_int` A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
4885 \int_new:N \l__regex_show_lines_int
```

(End of definition for \l__regex_show_lines_int.)

50.3.2 Generic helpers used when compiling

`\__regex_two_if_eq:NNNNTF` Used to compare pairs of things like `\__regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behavior of `\if:w` is very useful as that means we can use `\c_left_brace_str` and the like.

```
4886 \cs_new:Npn \__regex_two_if_eq:NNNNTF #1#2#3#4
4887 {
4888   \if_meaning:w #1 #3
4889   \if:w #2 #4
4890   \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4891   \fi:
4892   \fi:
4893   \use_ii:nn
4894 }
```

(End of definition for __regex_two_if_eq:NNNNTF.)

`\__regex_get_digits:NTFw` If followed by some raw digits, collect them one by one in the integer variable #1, and
`\__regex_get_digits_loop:w` take the **true** branch. Otherwise, take the **false** branch.

```
4895 \cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
4896 {
4897   \__regex_if_raw_digit:NNTF #4 #5
4898   { #1 = #5 \__regex_get_digits_loop:nw {#2} }
4899   { #3 #4 #5 }
4900 }
4901 \cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
4902 {
4903   \__regex_if_raw_digit:NNTF #2 #3
4904   { #3 \__regex_get_digits_loop:nw {#1} }
4905   { \scan_stop: #1 #2 #3 }
4906 }
```

(End of definition for __regex_get_digits:NTFw and __regex_get_digits_loop:w.)

`\__regex_if_raw_digit:NNTF` Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
4907 \cs_new:Npn \__regex_if_raw_digit:NNTF #1#2
4908 {
4909   \if_meaning:w \__regex_compile_raw:N #1
4910   \if_int_compare:w \c_one_int < 1 #2 \exp_stop_f:
```

```

4911         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4912         \fi:
4913     \fi:
4914     \use_ii:nn
4915 }

```

(End of definition for `\_regex_if_raw_digit:NNTF`.)

50.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6 `[\c{...}]` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `[\c...]` catcode test in a class,
- 63 `[\c{[...]}]` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,
- 23 `\c[...]` class inside mode 2,
- 63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`\_regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```

4916 \prg_new_conditional:Npnn \_regex_if_in_class: { TF }
4917 {
4918   \if_int_odd:w \l__regex_mode_int
4919   \prg_return_true:
4920   \else:
4921     \prg_return_false:
4922   \fi:
4923 }

```

(End of definition for _regex_if_in_class:TF.)

`\_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```

4924 \cs_new:Npn \_regex_if_in_cs:TF
4925 {
4926   \if_int_odd:w \l__regex_mode_int
4927   \else:
4928     \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
4929     \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
4930   \fi:
4931   \fi:
4932   \use_ii:nn
4933 }

```

(End of definition for _regex_if_in_cs:TF.)

`\_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, i.e., even, non-positive modes.

```

4934 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
4935 {
4936   \if_int_odd:w \l__regex_mode_int
4937   \else:
4938     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4939     \else:
4940       \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
4941     \fi:
4942   \fi:
4943   \use_i:nn
4944 }

```

(End of definition for _regex_if_in_class_or_catcode:TF.)

`\_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```

4945 \prg_new_conditional:Npnn \_regex_if_within_catcode: { TF }
4946 {
4947   \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
4948   \prg_return_true:
4949   \else:
4950     \prg_return_false:
4951   \fi:
4952 }

```

(End of definition for `\__regex_if_within_catcode:TF`.)

`\__regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, i.e., not within any other `\c` escape sequence.

```

4953 \cs_new_protected:Npn \__regex_chk_c_allowed:T
4954 {
4955   \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
4956   \else:
4957     \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
4958     \else:
4959       \msg_error:nn { regex } { c-bad-mode }
4960       \exp_after:wN \use_i:nnn
4961     \fi:
4962   \fi:
4963   \use:n
4964 }

```

(End of definition for `\__regex_chk_c_allowed:T`.)

`\__regex_mode_quit:c:` This function changes the mode as it is needed just after a catcode test.

```

4965 \cs_new_protected:Npn \__regex_mode_quit:c:
4966 {
4967   \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
4968   \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
4969   \else:
4970     \if_int_compare:w \l__regex_mode_int =
4971       \c__regex_catcode_in_class_mode_int
4972     \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
4973   \fi:
4974   \fi:
4975 }

```

(End of definition for `\__regex_mode_quit:c:`.)

50.3.4 Framework

`\__regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within
`\__regex_compile_end:` another regex. Start building a token list within a group (with e-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_tmp_regex`.

```

4976 \cs_new_protected:Npn \__regex_compile:w
4977 {
4978   \group_begin:
4979     \tl_build_begin:N \l__regex_build_tl
4980     \int_zero:N \l__regex_group_level_int
4981     \int_set_eq:NN \l__regex_default_catcodes_int
4982       \c__regex_all_catcodes_int
4983     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
4984     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
4985     \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
4986     \tl_build_put_right:Nn \l__regex_build_tl
4987       { \__regex_branch:n { \if_false: } \fi: }
4988 }

```

```

4989 \cs_new_protected:Npn \__regex_compile_end:
4990 {
4991   \__regex_if_in_class:TF
4992   {
4993     \msg_error:nn { regex } { missing-rbrack }
4994     \use:c { __regex_compile_]: }
4995     \prg_do_nothing: \prg_do_nothing:
4996   }
4997   { }
4998   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
4999   \msg_error:nne { regex } { missing-rparen }
5000   { \int_use:N \l__regex_group_level_int }
5001   \prg_replicate:nn
5002     \l__regex_group_level_int
5003     {
5004       \tl_build_put_right:Nn \l__regex_build_tl
5005       {
5006         \if_false: { \fi: }
5007         \if_false: { \fi: } { 1 } { 0 } \c_true_bool
5008       }
5009       \tl_build_end:N \l__regex_build_tl
5010       \exp_args:NNNo
5011       \group_end:
5012       \tl_build_put_right:Nn \l__regex_build_tl
5013       { \l__regex_build_tl }
5014     }
5015     \fi:
5016     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5017     \tl_build_end:N \l__regex_build_tl
5018     \exp_args:NNNe
5019     \group_end:
5020     \tl_set:Nn \l__regex_tmp_regex { \l__regex_build_tl }
5021   }

```

(End of definition for __regex_compile:w and __regex_compile_end:.)

__regex_compile:n The compilation is done between __regex_compile:w and __regex_compile_end:, starting in mode 0. Then __regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since __regex_compile_end: does that. However, catch the case of a trailing \cL construction.

```

5022 \cs_new_protected:Npn \__regex_compile:n #1
5023 {
5024   \__regex_compile:w
5025   \__regex_standard_escapechar:
5026   \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
5027   \__regex_escape_use:nnnn
5028   {
5029     \__regex_char_if_special:NTF ##1
5030     \__regex_compile_special:N \__regex_compile_raw:N ##1
5031   }
5032   {

```



```

5033     \__regex_char_if_alphanumeric:NTF ##1
5034     \__regex_compile_escaped:N \__regex_compile_raw:N ##1
5035   }
5036   { \__regex_compile_raw:N ##1 }
5037   { #1 }
5038   \prg_do_nothing: \prg_do_nothing:
5039   \prg_do_nothing: \prg_do_nothing:
5040   \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
5041   { \msg_error:nn { regex } { c-trailing } }
5042   \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
5043   {
5044     \msg_error:nn { regex } { c-missing-rbrace }
5045     \__regex_compile_end_cs:
5046     \prg_do_nothing: \prg_do_nothing:
5047     \prg_do_nothing: \prg_do_nothing:
5048   }
5049   \__regex_compile_end:
5050 }

```

(End of definition for __regex_compile:n.)

`\__regex_compile_use:n` Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```

5051 \cs_new_protected:Npn \__regex_compile_use:n #1
5052 {
5053   \tl_if_single_token:nT {#1}
5054   {
5055     \exp_after:wN \__regex_compile_use_aux:w
5056     \token_to_meaning:N #1 ~ \q__regex_nil
5057   }
5058   \__regex_compile:n {#1} \l__regex_tmp_regex
5059 }
5060 \cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
5061 {
5062   \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
5063   { \use_ii:nnn }
5064 }

```

(End of definition for __regex_compile_use:n.)

`\__regex_compile_escaped:N` If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

`\__regex_compile_special:N`

```

5065 \cs_new_protected:Npn \__regex_compile_special:N #1
5066 {
5067   \cs_if_exist_use:cF { __regex_compile_#1: }
5068   { \__regex_compile_raw:N #1 }
5069 }
5070 \cs_new_protected:Npn \__regex_compile_escaped:N #1
5071 {
5072   \cs_if_exist_use:cF { __regex_compile_/#1: }
5073   { \__regex_compile_raw:N #1 }
5074 }

```

(End of definition for `\__regex_compile_escaped:N` and `\__regex_compile_special:N`)

`\__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (e.g., `\cL`), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

5075 \cs_new_protected:Npn \__regex_compile_one:n #1
5076 {
5077   \__regex_mode_quit_c:
5078   \__regex_if_in_class:TF { }
5079   {
5080     \tl_build_put_right:Nn \l__regex_build_tl
5081     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5082   }
5083   \tl_build_put_right:Ne \l__regex_build_tl
5084   {
5085     \if_int_compare:w \l__regex_catcodes_int <
5086     \c__regex_all_catcodes_int
5087     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
5088     { \exp_not:N \exp_not:n {#1} }
5089     \else:
5090     \exp_not:N \exp_not:n {#1}
5091     \fi:
5092   }
5093   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5094   \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
5095 }

```

(End of definition for `\__regex_compile_one:n`)

`\__regex_compile_abort_tokens:n` This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```

5096 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
5097 {
5098   \use:e
5099   {
5100     \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
5101     \__regex_compile_raw:N
5102   }
5103 }
5104 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }

```

(End of definition for `\__regex_compile_abort_tokens:n`)

50.3.5 Quantifiers

`\__regex_compile_if_quantifier:TFw` This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{}`). This is useful for the `\u` and `\ur` escape sequences.

```

5105 \cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
5106 {
5107   \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
5108   { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
5109   { \use_ii:nn }
5110   {#1} {#2} #3 #4

```

```
5111 }
```

(End of definition for _regex_compile_if_quantifier:TFw.)

_regex_compile_quantifier:w This looks ahead and finds any quantifier (special character equal to either of ?+*{).

```
5112 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
5113 {
5114   \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5115   {
5116     \cs_if_exist_use:cF { \_regex_compile_quantifier_#2:w }
5117     { \_regex_compile_quantifier_none: #1 #2 }
5118   }
5119   { \_regex_compile_quantifier_none: #1 #2 }
5120 }
```

(End of definition for _regex_compile_quantifier:w.)

_regex_compile_quantifier_none: Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

_regex_compile_quantifier_abort:eNN

```
5121 \cs_new_protected:Npn \_regex_compile_quantifier_none:
5122 {
5123   \tl_build_put_right:Nn \l__regex_build_tl
5124   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
5125 }
5126 \cs_new_protected:Npn \_regex_compile_quantifier_abort:eNN #1#2#3
5127 {
5128   \_regex_compile_quantifier_none:
5129   \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
5130   \_regex_compile_abort_tokens:e {#1}
5131   #2 #3
5132 }
```

(End of definition for _regex_compile_quantifier_none: and _regex_compile_quantifier_abort:eNN.)

_regex_compile_quantifier_laziness:nnNN

Once the “main” quantifier (?, *, + or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending _regex_class:NnnnN and friends), the start-point of the range, its end-point, and a boolean, true for lazy and false for greedy operators.

```
5133 \cs_new_protected:Npn \_regex_compile_quantifier_laziness:nnNN #1#2#3#4
5134 {
5135   \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ?
5136   {
5137     \tl_build_put_right:Nn \l__regex_build_tl
5138     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
5139   }
5140   {
5141     \tl_build_put_right:Nn \l__regex_build_tl
5142     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
5143     #3 #4
5144   }
5145 }
```

(End of definition for _regex_compile_quantifier_laziness:nnNN.)

`\_regex_compile_quantifier?:w` For each “basic” quantifier, `?`, `*`, `+`, feed the correct arguments to `\_regex_compile_quantifier_laziness:nnNN`, `-1` means that there is no upper bound on the number of repetitions.

```
5146 \cs_new_protected:cpn { \_regex_compile_quantifier?:w }
5147   { \_regex_compile_quantifier_laziness:nnNN { 0 } { 1 } }
5148 \cs_new_protected:cpn { \_regex_compile_quantifier*:w }
5149   { \_regex_compile_quantifier_laziness:nnNN { 0 } { -1 } }
5150 \cs_new_protected:cpn { \_regex_compile_quantifier+:w }
5151   { \_regex_compile_quantifier_laziness:nnNN { 1 } { -1 } }
```

(End of definition for `\_regex_compile_quantifier?:w`, `\_regex_compile_quantifier*:w`, and `\_regex_compile_quantifier+:w`.)

`\_regex_compile_quantifier{w`
`\_regex_compile_quantifier_braced_auxi:w`
`\_regex_compile_quantifier_braced_auxii:w`
`\_regex_compile_quantifier_braced_auxiii:w`

Three possible syntaxes: `{⟨int⟩}`, `{⟨int⟩,}`, or `{⟨int⟩,⟨int⟩}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l__regex_tmpa_int`. If the number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as `{a}{-1}` and `{a}{b-a}`.

```
5152 \cs_new_protected:cpn { \_regex_compile_quantifier_ \c_left_brace_str :w }
5153   {
5154     \_regex_get_digits:NTFw \l__regex_tmpa_int
5155     { \_regex_compile_quantifier_braced_auxi:w }
5156     { \_regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
5157   }
5158 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxi:w #1#2
5159   {
5160     \str_case_e:nnF { #1 #2 }
5161     {
5162       { \_regex_compile_special:N \c_right_brace_str }
5163       {
5164         \exp_args:No \_regex_compile_quantifier_laziness:nnNN
5165           { \int_use:N \l__regex_tmpa_int } 0
5166       }
5167     { \_regex_compile_special:N , }
5168     {
5169       \_regex_get_digits:NTFw \l__regex_tmpb_int
5170       { \_regex_compile_quantifier_braced_auxiii:w }
5171       { \_regex_compile_quantifier_braced_auxii:w }
5172     }
5173   }
5174   {
5175     \_regex_compile_quantifier_abort:eNN
5176     { \c_left_brace_str \int_use:N \l__regex_tmpa_int }
5177     #1 #2
5178   }
5179 }
5180 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxii:w #1#2
5181   {
5182     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
5183     {
5184       \exp_args:No \_regex_compile_quantifier_laziness:nnNN
5185         { \int_use:N \l__regex_tmpa_int } { -1 }
5186     }
```

```

5187     {
5188         \_regex_compile_quantifier_abort:eNN
5189         { \c_left_brace_str \int_use:N \l__regex_tmpa_int , }
5190         #1 #2
5191     }
5192 }
5193 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxiii:w #1#2
5194 {
5195     \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_special:N \c_right_brace_str
5196     {
5197         \if_int_compare:w \l__regex_tmpa_int >
5198         \l__regex_tmpb_int
5199         \msg_error:nnee { regex } { backwards-quantifier }
5200         { \int_use:N \l__regex_tmpa_int }
5201         { \int_use:N \l__regex_tmpb_int }
5202         \int_zero:N \l__regex_tmpb_int
5203     }
5204     \else:
5205     \int_sub:Nn \l__regex_tmpb_int \l__regex_tmpa_int
5206     \fi:
5207     \exp_args:Noo \_regex_compile_quantifier_laziness:nnNN
5208     { \int_use:N \l__regex_tmpa_int }
5209     { \int_use:N \l__regex_tmpb_int }
5210 }
5211 {
5212     \_regex_compile_quantifier_abort:eNN
5213     {
5214         \c_left_brace_str
5215         \int_use:N \l__regex_tmpa_int ,
5216         \int_use:N \l__regex_tmpb_int
5217     }
5218     #1 #2
5219 }

```

(End of definition for _regex_compile_quantifier_{:w and others.)

50.3.6 Raw characters

_regex_compile_raw_error:N Within character classes, and following catcode tests, some escaped alphanumeric sequences such as \b do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

5220 \cs_new_protected:Npn \_regex_compile_raw_error:N #1
5221 {
5222     \msg_error:nne { regex } { bad-escape } {#1}
5223     \_regex_compile_raw:N #1
5224 }

```

(End of definition for _regex_compile_raw_error:N.)

_regex_compile_raw:N If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character #1 matches itself.

```

5225 \cs_new_protected:Npn \_regex_compile_raw:N #1#2#3
5226 {
5227     \_regex_if_in_class:TF

```

```

5228     {
5229         \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N -
5230         { \_regex_compile_range:Nw #1 }
5231         {
5232             \_regex_compile_one:n
5233             { \_regex_item_equal:n { \int_value:w '#1 } }
5234             #2 #3
5235         }
5236     }
5237     {
5238         \_regex_compile_one:n
5239         { \_regex_item_equal:n { \int_value:w '#1 } }
5240         #2 #3
5241     }
5242 }

```

(End of definition for _regex_compile_raw:N.)

_regex_compile_range:Nw
_regex_if_end_range:NNTF

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```

5243 \cs_new_protected:Npn \_regex_if_end_range:NNTF #1#2
5244 {
5245     \if_meaning:w \_regex_compile_raw:N #1
5246     \else:
5247         \if_meaning:w \_regex_compile_special:N #1
5248         \if_charcode:w ] #2
5249             \use_i:nn
5250         \fi:
5251     \else:
5252         \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
5253     \fi:
5254     \fi:
5255     \use_i:nn
5256 }
5257 \cs_new_protected:Npn \_regex_compile_range:Nw #1#2#3
5258 {
5259     \_regex_if_end_range:NNTF #2 #3
5260     {
5261         \if_int_compare:w '#1 > '#3 \exp_stop_f:
5262         \msg_error:nnee { regex } { range-backwards } {#1} {#3}
5263     \else:
5264         \tl_build_put_right:Ne \l__regex_build_tl
5265         {
5266             \if_int_compare:w '#1 = '#3 \exp_stop_f:
5267             \_regex_item_equal:n
5268             \else:
5269                 \_regex_item_range:nn { \int_value:w '#1 }
5270             \fi:
5271             { \int_value:w '#3 }
5272         }
5273     \fi:
5274 }
5275 {

```

```

5276     \msg_warning:nnee { regex } { range-missing-end }
5277     {#1} { \c_backslash_str #3 }
5278     \tl_build_put_right:Ne \l__regex_build_tl
5279     {
5280         \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
5281         \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
5282     }
5283     #2#3
5284 }
5285 }

```

(End of definition for `\__regex_compile_range:Nw` and `\__regex_if_end_range:NNTF`.)

50.3.7 Character properties

`\__regex_compile_.`: In a class, the dot has no special meaning. Outside, insert `\__regex_prop_.`, which matches any character or control sequence, and refuses `-2` (end-marker).

```

5286 \cs_new_protected:cpe { __regex_compile_.: }
5287 {
5288     \exp_not:N \__regex_if_in_class:TF
5289     { \__regex_compile_raw:N . }
5290     { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
5291 }
5292 \cs_new_protected:cpn { __regex_prop_.: }
5293 {
5294     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
5295     \exp_after:wN \__regex_break_true:w
5296     \fi:
5297 }

```

(End of definition for `\__regex_compile_.` and `\__regex_prop_.`.)

`\__regex_compile_/d:` The constants `\__regex_prop_d:`, etc. hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

5298 \cs_set_protected:Npn \__regex_tmp:w #1#2
5299 {
5300     \cs_new_protected:cpe { __regex_compile_/#1: }
5301     { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
5302     \cs_new_protected:cpe { __regex_compile_/#2: }
5303     {
5304         \__regex_compile_one:n
5305         { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
5306     }
5307 }
5308 \__regex_tmp:w d D
5309 \__regex_tmp:w h H
5310 \__regex_tmp:w s S
5311 \__regex_tmp:w v V
5312 \__regex_tmp:w w W
5313 \cs_new_protected:cpn { __regex_compile_/N: }
5314 { \__regex_compile_one:n \__regex_prop_N: }

```

(End of definition for `\__regex_compile_/d:` and others.)

50.3.8 Anchoring and simple assertions

`__regex_compile_anchor_letter:NNN` In modes where assertions are forbidden, anchors such as `\A` produce an error (`\A` is invalid in classes); otherwise they add an `__regex_assertion:Nn` test as appropriate (the only negative assertion is `\B`). The test functions are defined later. The implementation for `$` and `^` is only different from `\A` etc because these are valid in a class.

```

5315 \cs_new_protected:Npn __regex_compile_anchor_letter:NNN #1#2#3
5316 {
5317   __regex_if_in_class_or_catcode:TF { __regex_compile_raw_error:N #1 }
5318   {
5319     \tl_build_put_right:Nn \l__regex_build_tl
5320     { __regex_assertion:Nn #2 {#3} }
5321   }
5322 }
5323 \cs_new_protected:cpn { __regex_compile_/A: }
5324 { __regex_compile_anchor_letter:NNN A \c_true_bool __regex_A_test: }
5325 \cs_new_protected:cpn { __regex_compile_/G: }
5326 { __regex_compile_anchor_letter:NNN G \c_true_bool __regex_G_test: }
5327 \cs_new_protected:cpn { __regex_compile_/Z: }
5328 { __regex_compile_anchor_letter:NNN Z \c_true_bool __regex_Z_test: }
5329 \cs_new_protected:cpn { __regex_compile_/z: }
5330 { __regex_compile_anchor_letter:NNN z \c_true_bool __regex_Z_test: }
5331 \cs_new_protected:cpn { __regex_compile_/b: }
5332 { __regex_compile_anchor_letter:NNN b \c_true_bool __regex_b_test: }
5333 \cs_new_protected:cpn { __regex_compile_/B: }
5334 { __regex_compile_anchor_letter:NNN B \c_false_bool __regex_b_test: }
5335 \cs_set_protected:Npn __regex_tmp:w #1#2
5336 {
5337   \cs_new_protected:cpn { __regex_compile_#1: }
5338   {
5339     __regex_if_in_class_or_catcode:TF { __regex_compile_raw:N #1 }
5340     {
5341       \tl_build_put_right:Nn \l__regex_build_tl
5342       { __regex_assertion:Nn \c_true_bool {#2} }
5343     }
5344   }
5345 }
5346 \exp_args:Ne __regex_tmp:w { \iow_char:N ^ } { __regex_A_test: }
5347 \exp_args:Ne __regex_tmp:w { \iow_char:N $ } { __regex_Z_test: }

```

(End of definition for `__regex_compile_anchor_letter:NNN` and others.)

50.3.9 Character classes

`__regex_compile_]:` Outside a class, right brackets have no meaning. In a class, change the mode ($m \rightarrow (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[... \cL[...] ...]`). quantifiers.

```

5348 \cs_new_protected:cpn { __regex_compile_]: }
5349 {
5350   __regex_if_in_class:TF
5351   {
5352     \if_int_compare:w \l__regex_mode_int >
5353     \c_regex_catcode_in_class_mode_int
5354     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }

```



```

5355     \fi:
5356     \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
5357     \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
5358     \if_int_odd:w \l__regex_mode_int \else:
5359         \exp_after:wN \__regex_compile_quantifier:w
5360     \fi:
5361 }
5362 { \__regex_compile_raw:N ] }
5363 }

```

(End of definition for __regex_compile[:])

__regex_compile[: In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following \c<category>, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

5364 \cs_new_protected:cpn { \__regex_compile[: }
5365 {
5366     \__regex_if_in_class:TF
5367     { \__regex_compile_class_posix_test:w }
5368     {
5369         \__regex_if_within_catcode:TF
5370         {
5371             \exp_after:wN \__regex_compile_class_catcode:w
5372             \int_use:N \l__regex_catcodes_int \__regex_sep:
5373         }
5374         { \__regex_compile_class_normal:w }
5375     }
5376 }

```

(End of definition for __regex_compile[:])

__regex_compile_class_normal:w In the “normal” case, we insert __regex_class:NnnnN <boolean> in the compiled code. The <boolean> is true for positive classes, and false for negative classes, characterized by a leading ~. The auxiliary __regex_compile_class:TFNN also checks for a leading] which has a special meaning.

```

5377 \cs_new_protected:Npn \__regex_compile_class_normal:w
5378 {
5379     \__regex_compile_class:TFNN
5380     { \__regex_class:NnnnN \c_true_bool }
5381     { \__regex_class:NnnnN \c_false_bool }
5382 }

```

(End of definition for __regex_compile_class_normal:w.)

__regex_compile_class_catcode:w This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting __regex_item_catcode:nT or the reverse variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

5383 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1 \__regex_sep:
5384 {
5385     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5386         \tl_build_put_right:Nn \l__regex_build_tl

```

```

5387     { \_regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5388   \fi:
5389   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5390   \_regex_compile_class:TFNN
5391     { \_regex_item_catcode:nT {#1} }
5392     { \_regex_item_catcode_reverse:nT {#1} }
5393   }

```

(End of definition for _regex_compile_class_catcode:w.)

_regex_compile_class:TFNN If the first character is ^, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```

5394 \cs_new_protected:Npn \_regex_compile_class:TFNN #1#2#3#4
5395 {
5396   \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
5397   \_regex_two_if_eq:NNNTF #3 #4 \_regex_compile_special:N ^
5398   {
5399     \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
5400     \_regex_compile_class:NN
5401   }
5402   {
5403     \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5404     \_regex_compile_class:NN #3 #4
5405   }
5406 }
5407 \cs_new_protected:Npn \_regex_compile_class:NN #1#2
5408 {
5409   \token_if_eq_charcode:NNTF #2 ]
5410   { \_regex_compile_raw:N #2 }
5411   { #1 #2 }
5412 }

```

(End of definition for _regex_compile_class:TFNN and _regex_compile_class:NN.)

_regex_compile_class_posix_test:w Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra _regex_item_reverse:n for negative classes (we make sure to wrap its argument in braces otherwise \regex_show:N would not recognize the regex as valid).

```

5413 \cs_new_protected:Npn \_regex_compile_class_posix_test:w #1#2
5414 {
5415   \token_if_eq_meaning:NNT \_regex_compile_special:N #1
5416   {
5417     \str_case:nn { #2 }
5418     {
5419       : { \_regex_compile_class_posix:NNNNw }
5420       = {
5421         \msg_warning:nne { regex }
5422         { posix-unsupported } { = }
5423       }
5424       . {
5425         \msg_warning:nne { regex }
5426         { posix-unsupported } { . }

```

```

5427     }
5428   }
5429 }
5430 \__regex_compile_raw:N [ #1 #2
5431 }
5432 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
5433 {
5434   \__regex_two_if_eq:NNNTF #5 #6 \__regex_compile_special:N ^
5435   {
5436     \bool_set_false:N \l__regex_tmp_bool
5437     \__kernel_tl_set:Nx \l__regex_tmpa_tl { \if_false: } \fi:
5438     \__regex_compile_class_posix_loop:w
5439   }
5440   {
5441     \bool_set_true:N \l__regex_tmp_bool
5442     \__kernel_tl_set:Nx \l__regex_tmpa_tl { \if_false: } \fi:
5443     \__regex_compile_class_posix_loop:w #5 #6
5444   }
5445 }
5446 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
5447 {
5448   \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
5449   { #2 \__regex_compile_class_posix_loop:w }
5450   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
5451 }
5452 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
5453 {
5454   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N :
5455   { \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ] }
5456   { \use_ii:nn }
5457   {
5458     \cs_if_exist:cTF { __regex_posix_ \l__regex_tmpa_tl : }
5459     {
5460       \__regex_compile_one:n
5461       {
5462         \bool_if:NNTF \l__regex_tmp_bool \use:n \__regex_item_reverse:n
5463         { \exp_not:c { __regex_posix_ \l__regex_tmpa_tl : } }
5464       }
5465     }
5466     {
5467       \msg_warning:nne { regex } { posix-unknown }
5468       { \l__regex_tmpa_tl }
5469       \__regex_compile_abort_tokens:e
5470       {
5471         [: \bool_if:NNTF \l__regex_tmp_bool { ^ }
5472         \l__regex_tmpa_tl :]
5473       }
5474     }
5475   }
5476   {
5477     \msg_error:nnee { regex } { posix-missing-close }
5478     { [: \l__regex_tmpa_tl } { #2 #4 }
5479     \__regex_compile_abort_tokens:e { [: \l__regex_tmpa_tl }
5480     #1 #2 #3 #4

```

```

5481     }
5482 }

```

(End of definition for `\_regex_compile_class_posix_test:w` and others.)

50.3.10 Groups and alternations

```

\_regex_compile_group_begin:N
\_regex_compile_group_end:

```

The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `\_regex_branch:n {⟨concatenation⟩}`. This construction is done using `\tl_build_...` functions within a T_EX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is `\_regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```

5483 \cs_new_protected:Npn \_regex_compile_group_begin:N #1
5484 {
5485   \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
5486   \_regex_mode_quit_c:
5487   \group_begin:
5488     \tl_build_begin:N \l__regex_build_tl
5489     \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
5490     \int_incr:N \l__regex_group_level_int
5491     \tl_build_put_right:Nn \l__regex_build_tl
5492       { \_regex_branch:n { \if_false: } \fi: }
5493   }
5494 \cs_new_protected:Npn \_regex_compile_group_end:
5495 {
5496   \if_int_compare:w \l__regex_group_level_int > \c_zero_int
5497     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5498     \tl_build_end:N \l__regex_build_tl
5499     \exp_args:NNNe
5500     \group_end:
5501     \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
5502     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5503     \exp_after:wN \_regex_compile_quantifier:w
5504   \else:
5505     \msg_warning:nn { regex } { extra-rparen }
5506     \exp_after:wN \_regex_compile_raw:N \exp_after:wN )
5507   \fi:
5508 }

```

(End of definition for `\_regex_compile_group_begin:N` and `\_regex_compile_group_end:.`)

```

\_regex_compile(:

```

In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a `?`, denoting special groups, and run the code for the corresponding special group.

```

5509 \cs_new_protected:cpn { \_regex_compile(: }
5510 {
5511   \_regex_if_in_class:TF { \_regex_compile_raw:N ( }
5512   {
5513     \if_int_compare:w \l__regex_mode_int =

```

```

5514         \c__regex_catcode_in_class_mode_int
5515         \msg_error:nn { regex } { c-lparen-in-class }
5516         \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
5517         \else:
5518         \exp_after:wN \__regex_compile_lparen:w
5519         \fi:
5520     }
5521 }
5522 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
5523 {
5524     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ?
5525     {
5526         \cs_if_exist_use:cF
5527         { __regex_compile_special_group\_token_to_str:N #4 :w }
5528         {
5529             \msg_warning:nne { regex } { special-group-unknown }
5530             { (? #4 }
5531             \__regex_compile_group_begin:N \__regex_group:nnnN
5532             \__regex_compile_raw:N ? #3 #4
5533         }
5534     }
5535     {
5536         \__regex_compile_group_begin:N \__regex_group:nnnN
5537         #1 #2 #3 #4
5538     }
5539 }

```

(End of definition for __regex_compile_(:))

`\__regex_compile_|`: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

5540 \cs_new_protected:cpn { __regex_compile_|: }
5541 {
5542     \__regex_if_in_class:TF { \__regex_compile_raw:N | }
5543     {
5544         \tl_build_put_right:Nn \l__regex_build_tl
5545         { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
5546     }
5547 }

```

(End of definition for __regex_compile_|(:))

`\__regex_compile_)`: Within a class, parentheses are not special. Outside, close a group.

```

5548 \cs_new_protected:cpn { __regex_compile_): }
5549 {
5550     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
5551     { \__regex_compile_group_end: }
5552 }

```

(End of definition for __regex_compile_)(:))

`\_regex_compile_special_group::w` `\_regex_compile_special_group_|:w` Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

5553 \cs_new_protected:cpn { __regex_compile_special_group::w }
5554 { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }

```

```

5555 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
5556 { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }

(End of definition for \__regex_compile_special_group_:w and \__regex_compile_special_group_|:w.)

```

`\__regex_compile_special_group_i:w`
`\__regex_compile_special_group_-:w`

The match can be made case-insensitive by setting the option with `(?i)`; the original behavior is restored by `(?-i)`. This is the only supported option.

```

5557 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
5558 {
5559   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N )
5560   {
5561     \cs_set:Npn \__regex_item_equal:n
5562     { \__regex_item_caseless_equal:n }
5563     \cs_set:Npn \__regex_item_range:nn
5564     { \__regex_item_caseless_range:nn }
5565   }
5566   {
5567     \msg_warning:nne { regex } { unknown-option } { (?i #2 }
5568     \__regex_compile_raw:N (
5569     \__regex_compile_raw:N ?
5570     \__regex_compile_raw:N i
5571     #1 #2
5572   }
5573 }
5574 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
5575 {
5576   \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_raw:N i
5577   { \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ) }
5578   { \use_ii:nn }
5579   {
5580     \cs_set:Npn \__regex_item_equal:n
5581     { \__regex_item_caseful_equal:n }
5582     \cs_set:Npn \__regex_item_range:nn
5583     { \__regex_item_caseful_range:nn }
5584   }
5585   {
5586     \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
5587     \__regex_compile_raw:N (
5588     \__regex_compile_raw:N ?
5589     \__regex_compile_raw:N -
5590     #1 #2 #3 #4
5591   }
5592 }

```

(End of definition for `\__regex_compile_special_group_i:w` and `\__regex_compile_special_group_-:w`.)

50.3.11 Catcodes and csnames

`\__regex_compile_/c:`
`\__regex_compile_c_test:NN`

The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

5593 \cs_new_protected:cpn { __regex_compile_/c: }

```

```

5594 { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
5595 \cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
5596 {
5597   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5598   {
5599     \int_if_exist:cTF { c__regex_catcode_#2_int }
5600     {
5601       \int_set_eq:Nc \l__regex_catcodes_int
5602       { c__regex_catcode_#2_int }
5603       \l__regex_mode_int
5604       = \if_case:w \l__regex_mode_int
5605       \c__regex_catcode_mode_int
5606       \else:
5607       \c__regex_catcode_in_class_mode_int
5608       \fi:
5609       \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
5610     }
5611   }
5612   { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
5613   {
5614     \msg_error:nne { regex } { c-missing-category } {#2}
5615     #1 #2
5616   }
5617 }

```

(End of definition for __regex_compile_/c: and __regex_compile_c_test:NN.)

__regex_compile_c_C:NN If \cC is not followed by . or (...) then complain because that construction cannot match anything, except in cases like \cC[\c{...}], where it has no effect.

```

5618 \cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
5619 {
5620   \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5621   {
5622     \token_if_eq_charcode:NNTF #2 .
5623     { \use_none:n }
5624     { \token_if_eq_charcode:NNTF #2 ( } % )
5625   }
5626   { \use:n }
5627   { \msg_error:nnn { regex } { c-C-invalid } {#2} }
5628   #1 #2
5629 }

```

(End of definition for __regex_compile_c_C:NN.)

__regex_compile_c[:w] When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

\__regex_compile_c_lbrack_loop:NN
\__regex_compile_c_lbrack_add:N
\__regex_compile_c_lbrack_end:
5630 \cs_new_protected:cpn { __regex_compile_c[:w] } #1#2
5631 {
5632   \l__regex_mode_int
5633   = \if_case:w \l__regex_mode_int
5634   \c__regex_catcode_mode_int
5635   \else:
5636   \c__regex_catcode_in_class_mode_int
5637   \fi:

```

```

5638 \int_zero:N \l__regex_catcodes_int
5639 \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ^
5640 {
5641   \bool_set_false:N \l__regex_catcodes_bool
5642   \__regex_compile_c_lbrack_loop:NN
5643 }
5644 {
5645   \bool_set_true:N \l__regex_catcodes_bool
5646   \__regex_compile_c_lbrack_loop:NN
5647   #1 #2
5648 }
5649 }
5650 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
5651 {
5652   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5653   {
5654     \int_if_exist:cTF { c__regex_catcode_#2_int }
5655     {
5656       \exp_args:Nc \__regex_compile_c_lbrack_add:N
5657       { c__regex_catcode_#2_int }
5658       \__regex_compile_c_lbrack_loop:NN
5659     }
5660   }
5661   {
5662     \token_if_eq_charcode:NNTF #2 ]
5663     { \__regex_compile_c_lbrack_end: }
5664   }
5665   {
5666     \msg_error:nne { regex } { c-missing-rbrack } {#2}
5667     \__regex_compile_c_lbrack_end:
5668     #1 #2
5669   }
5670 }
5671 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
5672 {
5673   \if_int_odd:w \__regex_int_eval:w \l__regex_catcodes_int / #1 \scan_stop:
5674   \else:
5675     \int_add:Nn \l__regex_catcodes_int {#1}
5676   \fi:
5677 }
5678 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:
5679 {
5680   \if_meaning:w \c_false_bool \l__regex_catcodes_bool
5681   \int_set:Nn \l__regex_catcodes_int
5682   { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
5683   \fi:
5684 }

```

(End of definition for __regex_compile_c[:w and others.)

__regex_compile_c_{: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

5685 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }

```



```

5686 {
5687   \__regex_compile:w
5688   \__regex_disable_submatches:
5689   \l__regex_mode_int
5690   = \if_case:w \l__regex_mode_int
5691     \c__regex_cs_mode_int
5692   \else:
5693     \c__regex_cs_in_class_mode_int
5694   \fi:
5695 }

```

(End of definition for __regex_compile_c{:.})

__regex_compile_{: We forbid unescaped left braces inside a \c{...} escape because they otherwise lead to the confusing question of whether the first right brace in \c{{}x} should end \c or whether one should match braces.

```

5696 \cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
5697 {
5698   \__regex_if_in_cs:TF
5699   { \msg_error:nnn { regex } { cu-lbrace } { c } }
5700   { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
5701 }

```

(End of definition for __regex_compile_{:.})

\l__regex_cs_flag Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use __regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```

5702 \flag_new:N \l__regex_cs_flag
5703 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
5704 {
5705   \__regex_if_in_cs:TF
5706   { \__regex_compile_end_cs: }
5707   { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
5708 }
5709 \cs_new_protected:Npn \__regex_compile_end_cs:
5710 {
5711   \__regex_compile_end:
5712   \flag_clear:N \l__regex_cs_flag
5713   \__kernel_tl_set:Nx \l__regex_tmpa_tl
5714   {
5715     \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_tmp_regex
5716     \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5717   }
5718   \exp_args:Ne \__regex_compile_one:n
5719   {
5720     \flag_if_raised:NTF \l__regex_cs_flag
5721     { \__regex_item_cs:n { \exp_not:o \l__regex_tmp_regex } }
5722     {
5723       \__regex_item_exact_cs:n

```

```

5724         { \tl_tail:N \l__regex_tmpa_tl }
5725     }
5726 }
5727 }
5728 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
5729 {
5730     \cs_if_eq:NNTF #1 \__regex_branch:n
5731     {
5732         \scan_stop:
5733         \__regex_compile_cs_aux:NNnnN #2
5734         \q__regex_nil \q__regex_nil \q__regex_nil
5735         \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5736         \__regex_compile_cs_aux:Nn
5737     }
5738     {
5739         \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:N \l__regex_cs_flag }
5740         \__regex_use_none_delimit_by_q_recursion_stop:w
5741     }
5742 }
5743 \cs_new:Npn \__regex_compile_cs_aux:NNnnN #1#2#3#4#5#6
5744 {
5745     \bool_lazy_all:nTF
5746     {
5747         { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
5748         {#2}
5749         { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
5750         { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
5751         { \int_compare_p:nNn {#5} = \c_zero_int }
5752     }
5753     {
5754         \prg_replicate:nn {#4}
5755         { \char_generate:nn { \use_ii:nn #3 } {12} }
5756         \__regex_compile_cs_aux:NNnnN
5757     }
5758     {
5759         \__regex_quark_if_nil:NF #1
5760         {
5761             \flag_ensure_raised:N \l__regex_cs_flag
5762             \__regex_use_i_delimit_by_q_recursion_stop:nw
5763         }
5764         \__regex_use_none_delimit_by_q_recursion_stop:w
5765     }
5766 }

```

(End of definition for \l__regex_cs_flag and others.)

50.3.12 Raw token lists with \u

__regex_compile_/u: The \u escape is invalid in classes and directly following a catcode test. Otherwise test for a following r (for \ur), and call an auxiliary responsible for finding the variable name.

```

5767 \cs_new_protected:cpn { __regex_compile_/u: } #1#2
5768 {
5769     \__regex_if_in_class_or_catcode:TF
5770     { \__regex_compile_raw_error:N u #1 #2 }

```

```

5771     {
5772         \_regex_two_if_eq:NNNTF #1 #2 \_regex_compile_raw:N r
5773         { \_regex_compile_u_brace:NNN \_regex_compile_ur_end: }
5774         { \_regex_compile_u_brace:NNN \_regex_compile_u_end: #1 #2 }
5775     }
5776 }

```

(End of definition for _regex_compile_/u:.)

_regex_compile_u_brace:NNN This enforces the presence of a left brace, then starts a loop to find the variable name.

```

5777 \cs_new:Npn \_regex_compile_u_brace:NNN #1#2#3
5778 {
5779     \_regex_two_if_eq:NNNTF #2 #3 \_regex_compile_special:N \c_left_brace_str
5780     {
5781         \tl_set:Nn \l__regex_tmpb_tl {#1}
5782         \_kernel_tl_set:Nx \l__regex_tmpa_tl { \if_false: } \fi:
5783         \_regex_compile_u_loop:NN
5784     }
5785     {
5786         \msg_error:nn { regex } { u-missing-lbrace }
5787         \token_if_eq_meaning:NNTF #1 \_regex_compile_ur_end:
5788         { \_regex_compile_raw:N u \_regex_compile_raw:N r }
5789         { \_regex_compile_raw:N u }
5790         #2 #3
5791     }
5792 }

```

(End of definition for _regex_compile_u_brace:NNN.)

_regex_compile_u_loop:NN We collect the characters for the argument of \u within an e-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

5793 \cs_new:Npn \_regex_compile_u_loop:NN #1#2
5794 {
5795     \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
5796     { #2 \_regex_compile_u_loop:NN }
5797     {
5798         \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5799         {
5800             \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
5801             { \if_false: { \fi: } \l__regex_tmpb_tl }
5802             {
5803                 \if_charcode:w \c_left_brace_str #2
5804                 \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
5805                 \else:
5806                     #2
5807                 \fi:
5808                 \_regex_compile_u_loop:NN
5809             }
5810         }
5811         {
5812             \if_false: { \fi: }

```

```

5813         \msg_error:nne { regex } { u-missing-rbrace } {#2}
5814         \l__regex_tmpb_tl
5815         #1 #2
5816     }
5817 }
5818 }

```

(End of definition for __regex_compile_u_loop:NN.)

__regex_compile_ur_end: For the \ur{...} construction, once we have extracted the variable's name, we replace all groups by non-capturing groups in the compiled regex (passed as the argument of __regex_compile_ur:n). If that has a single branch (namely \tl_if_empty:oTF is false) and there is no quantifier, then simply insert the contents of this branch (obtained by \use_ii:nn, which is expanded later). In all other cases, insert a non-capturing group and look for quantifiers to determine the number of repetition etc.

```

5819 \cs_new_protected:Npn \__regex_compile_ur_end:
5820 {
5821     \group_begin:
5822     \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
5823     \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
5824     \exp_args:NNe
5825     \group_end:
5826     \__regex_compile_ur:n { \use:c { \l__regex_tmpa_tl } }
5827 }
5828 \cs_new_protected:Npn \__regex_compile_ur:n #1
5829 {
5830     \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} } ? ? \q__regex_nil }
5831     { \__regex_compile_if_quantifier:TFw }
5832     { \use_i:nn }
5833     {
5834         \tl_build_put_right:Nn \l__regex_build_tl
5835         { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
5836         \__regex_compile_quantifier:w
5837     }
5838     { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
5839 }
5840 \cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}

```

(End of definition for __regex_compile_ur_end:, __regex_compile_ur:n, and __regex_compile_ur_aux:w.)

__regex_compile_u_end: Once we have extracted the variable's name, we check for quantifiers, in which case we set up a non-capturing group with a single branch. Inside this branch (we omit it and the group if there is no quantifier), __regex_compile_u_payload: puts the right tests corresponding to the contents of the variable, which we store in \l__regex_tmpa_tl. The behavior of \u then depends on whether we are within a \c{...} escape (in this case, the variable is turned to a string), or not.

```

5841 \cs_new_protected:Npn \__regex_compile_u_end:
5842 {
5843     \__regex_compile_if_quantifier:TFw
5844     {
5845         \tl_build_put_right:Nn \l__regex_build_tl
5846         {
5847             \__regex_group_no_capture:nnnN { \if_false: } \fi:

```

```

5848         \_regex_branch:n { \if_false: } \fi:
5849     }
5850     \_regex_compile_u_payload:
5851     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5852     \_regex_compile_quantifier:w
5853 }
5854 { \_regex_compile_u_payload: }
5855 }
5856 \cs_new_protected:Npn \_regex_compile_u_payload:
5857 {
5858     \tl_set:Nv \l__regex_tmpa_tl { \l__regex_tmpa_tl }
5859     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
5860         \_regex_compile_u_not_cs:
5861     \else:
5862         \_regex_compile_u_in_cs:
5863     \fi:
5864 }

```

(End of definition for _regex_compile_u_end: and _regex_compile_u_payload:.)

_regex_compile_u_in_cs: When \u appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

5865 \cs_new_protected:Npn \_regex_compile_u_in_cs:
5866 {
5867     \_kernel_tl_gset:Nx \g__regex_tmp_tl
5868     {
5869         \exp_args:No \_kernel_str_to_other_fast:n
5870         { \l__regex_tmpa_tl }
5871     }
5872     \tl_build_put_right:Ne \l__regex_build_tl
5873     {
5874         \tl_map_function:NN \g__regex_tmp_tl
5875         \_regex_compile_u_in_cs_aux:n
5876     }
5877 }
5878 \cs_new:Npn \_regex_compile_u_in_cs_aux:n #1
5879 {
5880     \_regex_class:NnnN \c_true_bool
5881     { \_regex_item_caseful_equal:n { \int_value:w '#1 } }
5882     { 1 } { 0 } \c_false_bool
5883 }

```

(End of definition for _regex_compile_u_in_cs:.)

_regex_compile_u_not_cs: In mode 0, the \u escape adds one state to the NFA for each token in \l__regex_tmpa_tl. If a given *<token>* is a control sequence, then insert a string comparison test, otherwise, _regex_item_exact:nn which compares catcode and character code.

```

5884 \cs_new_protected:Npn \_regex_compile_u_not_cs:
5885 {
5886     \tl_analysis_map_inline:Nn \l__regex_tmpa_tl
5887     {
5888         \tl_build_put_right:Ne \l__regex_build_tl
5889         {

```

```

5890         \__regex_class:NnnnN \c_true_bool
5891         {
5892             \if_int_compare:w "##3 = \c_zero_int
5893             \__regex_item_exact_cs:n
5894             { \exp_after:wN \cs_to_str:N ##1 }
5895             \else:
5896             \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
5897             \fi:
5898         }
5899         { 1 } { 0 } \c_false_bool
5900     }
5901 }
5902 }

```

(End of definition for __regex_compile_u_not_cs:.)

50.3.13 Other

__regex_compile_/K: The \K control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as \b. At the compilation stage, we leave it as a single control sequence, defined later.

```

5903 \cs_new_protected:cpn { __regex_compile_/K: }
5904 {
5905     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
5906     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
5907     { \__regex_compile_raw_error:N K }
5908 }

```

(End of definition for __regex_compile_/K:.)

50.3.14 Showing regexes

Before showing a regex we check that it is “clean” in the sense that it has the correct internal structure. We do this (in the implementation of \regex_show:N and \regex_log:N) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all __regex_clean_⟨type⟩:n functions produce valid ⟨type⟩ tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```

5909 \cs_new:Npn \__regex_clean_bool:n #1
5910 {
5911     \tl_if_single:nTF {#1}
5912     { \bool_if:NTF #1 \c_true_bool \c_false_bool }
5913     { \c_true_bool }
5914 }
5915 \cs_new:Npn \__regex_clean_int:n #1
5916 {
5917     \tl_if_head_eq_meaning:nNTF {#1} -
5918     { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
5919     { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
5920 }
5921 \cs_new:Npn \__regex_clean_int_aux:N #1
5922 {
5923     \if_int_compare:w \c_one_int < 1 #1 ~

```

```

5924         #1
5925     \else:
5926         \str_map_break:n
5927     \fi:
5928 }
5929 \cs_new:Npn \__regex_clean_regex:n #1
5930 {
5931     \__regex_clean_regex_loop:w #1
5932     \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
5933 }
5934 \cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
5935 {
5936     \quark_if_recursion_tail_stop:n {#2}
5937     \__regex_branch:n { \__regex_clean_branch:n {#2} }
5938     \__regex_clean_regex_loop:w
5939 }
5940 \cs_new:Npn \__regex_clean_branch:n #1
5941 {
5942     \__regex_clean_branch_loop:n #1
5943     ? ? ? ? ? \prg_break_point:
5944 }
5945 \cs_new:Npn \__regex_clean_branch_loop:n #1
5946 {
5947     \tl_if_single:nF {#1} \prg_break:
5948     \token_case_meaning:NnF #1
5949     {
5950         \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
5951         \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
5952         \__regex_class:NnnN { #1 \__regex_clean_class:NnnN }
5953         \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
5954         \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
5955         \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
5956     }
5957     \prg_break:
5958 }
5959 \cs_new:Npn \__regex_clean_assertion:Nn #1#2
5960 {
5961     \__regex_clean_bool:n {#1}
5962     \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
5963     \token_case_meaning:NnTF #2
5964     {
5965         \__regex_A_test: { }
5966         \__regex_G_test: { }
5967         \__regex_Z_test: { }
5968         \__regex_b_test: { }
5969     }
5970     { {#2} }
5971     { { \__regex_A_test: } \prg_break: }
5972     \__regex_clean_branch_loop:n
5973 }
5974 \cs_new:Npn \__regex_clean_class:NnnN #1#2#3#4#5
5975 {
5976     \__regex_clean_bool:n {#1}
5977     { \__regex_clean_class:n {#2} }

```

```

5978     { \int_max:nn \c_zero_int { \__regex_clean_int:n {#3} } }
5979     { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#4} } }
5980     \__regex_clean_bool:n {#5}
5981     \__regex_clean_branch_loop:n
5982   }
5983 \cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
5984 {
5985   { \__regex_clean_regex:n {#1} }
5986   { \int_max:nn \c_zero_int { \__regex_clean_int:n {#2} } }
5987   { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#3} } }
5988   \__regex_clean_bool:n {#4}
5989   \__regex_clean_branch_loop:n
5990 }
5991 \cs_new:Npn \__regex_clean_class:n #1
5992 { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }

```

When cleaning a class there are many cases, including a dozen or so like `\__regex_prop_d:` or `\__regex_posix_alpha:`. To avoid listing all of them we allow any command that starts with the 13 characters `__regex_prop_` or `__regex_posix` (handily these have the same length, except for the trailing underscore).

```

5993 \cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
5994 {
5995   \tl_if_single:nF {#1} \prg_break:
5996   \token_case_meaning:NnTF #1
5997   {
5998     \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }
5999     \__regex_item_exact_cs:n { #1 { \__regex_clean_exact_cs:n {#2} } }
6000     \__regex_item_caseful_equal:n { #1 { \__regex_clean_int:n {#2} } }
6001     \__regex_item_caseless_equal:n { #1 { \__regex_clean_int:n {#2} } }
6002     \__regex_item_reverse:n { #1 { \__regex_clean_class:n {#2} } }
6003   }
6004   { \__regex_clean_class_loop:nnn {#3} }
6005   {
6006     \token_case_meaning:NnTF #1
6007     {
6008       \__regex_item_caseful_range:nn { }
6009       \__regex_item_caseless_range:nn { }
6010       \__regex_item_exact:nn { }
6011     }
6012     {
6013       #1 { \__regex_clean_int:n {#2} } { \__regex_clean_int:n {#3} }
6014       \__regex_clean_class_loop:nnn
6015     }
6016     {
6017       \token_case_meaning:NnTF #1
6018       {
6019         \__regex_item_catcode:nT { }
6020         \__regex_item_catcode_reverse:nT { }
6021       }
6022       {
6023         #1 { \__regex_clean_int:n {#2} } { \__regex_clean_class:n {#3} }
6024         \__regex_clean_class_loop:nnn
6025       }
6026     }

```



```

6027         \exp_args:Ne \str_case:nnTF
6028         {
6029             \exp_args:Ne \str_range:nnn
6030             { \cs_to_str:N #1 } \c_one_int { 13 }
6031         }
6032         {
6033             { __regex_prop_ } { }
6034             { __regex_posix } { }
6035         }
6036         {
6037             #1
6038             \__regex_clean_class_loop:nnn {#2} {#3}
6039         }
6040         \prg_break:
6041     }
6042 }
6043 }
6044 }
6045 \cs_new:Npn \__regex_clean_exact_cs:n #1
6046 {
6047     \exp_last_unbraced:Nf \use_none:n
6048     {
6049         \__regex_clean_exact_cs:w #1
6050         \scan_stop: \q_recursion_tail \scan_stop:
6051         \q_recursion_stop
6052     }
6053 }
6054 \cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
6055 {
6056     \quark_if_recursion_tail_stop:n {#1}
6057     \scan_stop: \tl_to_str:n {#1}
6058     \__regex_clean_exact_cs:w
6059 }

```

(End of definition for __regex_clean_bool:n and others.)

__regex_show:N Within a group and within \tl_build_begin:N ... \tl_build_end:N we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in \l__regex_tmpa_tl is then meant to be shown.

```

6060 \cs_new_protected:Npn \__regex_show:N #1
6061 {
6062     \group_begin:
6063     \tl_build_begin:N \l__regex_build_tl
6064     \cs_set_protected:Npn \__regex_branch:n
6065     {
6066         \seq_pop_right:NN \l__regex_show_prefix_seq
6067         \l__regex_tmpa_tl
6068         \__regex_show_one:n { +-branch }
6069         \seq_put_right:No \l__regex_show_prefix_seq
6070         \l__regex_tmpa_tl
6071         \use:n
6072     }
6073     \cs_set_protected:Npn \__regex_group:nnnN
6074     { \__regex_show_group_aux:nnnnN { } }

```

```

6075 \cs_set_protected:Npn \__regex_group_no_capture:nnnN
6076 { \__regex_show_group_aux:nnnnN { ~(no-capture) } }
6077 \cs_set_protected:Npn \__regex_group_resetting:nnnN
6078 { \__regex_show_group_aux:nnnnN { ~(resetting) } }
6079 \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
6080 \cs_set_protected:Npn \__regex_command_K:
6081 { \__regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
6082 \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
6083 {
6084   \__regex_show_one:n
6085   { \bool_if:NF ##1 { negative~ } assertion:~##2 }
6086 }
6087 \cs_set:Npn \__regex_b_test: { word-boundary }
6088 \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\Z) }
6089 \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\A) }
6090 \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\G) }
6091 \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
6092 { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
6093 \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
6094 {
6095   \__regex_show_one:n
6096   { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
6097 }
6098 \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
6099 { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
6100 \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
6101 {
6102   \__regex_show_one:n
6103   { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
6104 }
6105 \cs_set_protected:Npn \__regex_item_catcode:nT
6106 { \__regex_show_item_catcode:NnT \c_true_bool }
6107 \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
6108 { \__regex_show_item_catcode:NnT \c_false_bool }
6109 \cs_set_protected:Npn \__regex_item_reverse:n
6110 { \__regex_show_scope:nn { Reversed~match } }
6111 \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
6112 { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
6113 \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
6114 \cs_set_protected:Npn \__regex_item_cs:n
6115 { \__regex_show_scope:nn { control~sequence } }
6116 \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any-token } }
6117 \seq_clear:N \l__regex_show_prefix_seq
6118 \__regex_show_push:n { ~ }
6119 \cs_if_exist_use:N #1
6120 \tl_build_end:N \l__regex_build_tl
6121 \exp_args:NNNo
6122 \group_end:
6123 \tl_set:Nn \l__regex_tmpa_tl { \l__regex_build_tl }
6124 }

```

(End of definition for __regex_show:N.)

__regex_show_char:n Show a single character, together with its ascii representation if available. This could be

extended to beyond ascii. It is not ideal for parentheses themselves.

```

6125 \cs_new:Npn \__regex_show_char:n #1
6126 {
6127   \int_eval:n {#1}
6128   \int_compare:nT { 32 <= #1 <= 126 }
6129   { ~ ( \char_generate:nn {#1} {12} ) }
6130 }

```

(End of definition for __regex_show_char:n.)

`\__regex_show_one:n` Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

6131 \cs_new_protected:Npn \__regex_show_one:n #1
6132 {
6133   \int_incr:N \l__regex_show_lines_int
6134   \tl_build_put_right:Ne \l__regex_build_tl
6135   {
6136     \exp_not:N \iow_newline:
6137     \seq_map_function:NN \l__regex_show_prefix_seq \use:n
6138     #1
6139   }
6140 }

```

(End of definition for __regex_show_one:n.)

`\__regex_show_push:n` Enter and exit levels of nesting. The `scope` function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.
`\__regex_show_pop:`
`\__regex_show_scope:nn`

```

6141 \cs_new_protected:Npn \__regex_show_push:n #1
6142 { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
6143 \cs_new_protected:Npn \__regex_show_pop:
6144 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_tmpa_tl }
6145 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
6146 {
6147   \__regex_show_one:n {#1}
6148   \__regex_show_push:n { ~ }
6149   #2
6150   \__regex_show_pop:
6151 }

```

(End of definition for __regex_show_push:n, __regex_show_pop:, and __regex_show_scope:nn.)

`\__regex_show_group_aux:nnnnN` We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd `\use_ii:nn` avoids printing a spurious `+-branch` for the first branch.

```

6152 \cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
6153 {
6154   \__regex_show_one:n { , -group~begin #1 }
6155   \__regex_show_push:n { | }
6156   \use_ii:nn #2
6157   \__regex_show_pop:
6158   \__regex_show_one:n
6159   { ‘-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
6160 }

```

(End of definition for __regex_show_group_aux:nnnnN.)

`\__regex_show_class:NnnnN`

I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write `Match` or `Don't match` on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```
6161 \cs_new:Npn \__regex_show_class:NnnnN #1#2#3#4#5
6162 {
6163   \group_begin:
6164     \tl_build_begin:N \l__regex_build_tl
6165     \int_zero:N \l__regex_show_lines_int
6166     \__regex_show_push:n {~}
6167     #2
6168     \int_compare:nTF { \l__regex_show_lines_int = \c_zero_int }
6169     {
6170       \group_end:
6171       \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
6172     }
6173     {
6174       \bool_if:nTF
6175       { #1 && \int_compare_p:n { \l__regex_show_lines_int = \c_one_int } }
6176       {
6177         \group_end:
6178         #2
6179         \tl_build_put_right:Nn \l__regex_build_tl
6180         { \__regex_msg_repeated:nnN {#3} {#4} #5 }
6181       }
6182       {
6183         \tl_build_end:N \l__regex_build_tl
6184         \exp_args:NNNo
6185         \group_end:
6186         \tl_set:Nn \l__regex_tmpa_tl \l__regex_build_tl
6187         \__regex_show_one:n
6188         {
6189           \bool_if:NTF #1 { Match } { Don't-match }
6190           \__regex_msg_repeated:nnN {#3} {#4} #5
6191         }
6192         \tl_build_put_right:Ne \l__regex_build_tl
6193         { \exp_not:o \l__regex_tmpa_tl }
6194       }
6195     }
6196 }
```

(End of definition for `\__regex_show_class:NnnnN`.)

`\__regex_show_item_catcode:NnT`

Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```
6197 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
6198 {
6199   \seq_set_split:Nnn \l__regex_tmp_seq { } { CBEMTPUDSLOA }
6200   \seq_set_filter:NNn \l__regex_tmp_seq \l__regex_tmp_seq
6201   { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
6202   \__regex_show_scope:nn
6203   {
```

```

6204         categories~
6205         \seq_map_function:NN \l__regex_tmp_seq \use:n
6206         , ~
6207         \bool_if:NF #1 { negative~ } class
6208     }
6209 }

```

(End of definition for __regex_show_item_catcode:NnT.)

__regex_show_item_exact_cs:n

```

6210 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
6211 {
6212     \seq_set_split:Nnn \l__regex_tmp_seq { \scan_stop: } {#1}
6213     \seq_set_map_e:Nnn \l__regex_tmp_seq
6214         \l__regex_tmp_seq { \iow_char:N\##1 }
6215     \__regex_show_one:n
6216     { control~sequence~ \seq_use:Nn \l__regex_tmp_seq { ~or~ } }
6217 }

```

(End of definition for __regex_show_item_exact_cs:n.)

50.4 Building

50.4.1 Variables used while building

\l__regex_min_state_int The last state that was allocated is \l__regex_max_state_int − 1, so that \l__regex_max_state_int always points to a free state. The min_state variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in \c{...} constructions.

```

6218 \int_new:N \l__regex_min_state_int
6219 \int_set:Nn \l__regex_min_state_int { 1 }
6220 \int_new:N \l__regex_max_state_int

```

(End of definition for \l__regex_min_state_int and \l__regex_max_state_int.)

\l__regex_left_state_int Alternatives are implemented by branching from a left state into the various choices, then merging those into a right state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

6221 \int_new:N \l__regex_left_state_int
6222 \int_new:N \l__regex_right_state_int
6223 \seq_new:N \l__regex_left_state_seq
6224 \seq_new:N \l__regex_right_state_seq

```

(End of definition for \l__regex_left_state_int and others.)

\l__regex_capturing_group_int

\l__regex_capturing_group_int is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering resetting groups.

```

6225 \int_new:N \l__regex_capturing_group_int

```

(End of definition for \l__regex_capturing_group_int.)

50.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `\__regex_action_start_wildcard:N` $\langle \text{boolean} \rangle$ inserted at the start of the regular expression, where a `true` $\langle \text{boolean} \rangle$ makes it unanchored.
- `\__regex_action_success:` marks the exit state of the NFA.
- `\__regex_action_cost:n` $\{\langle \text{shift} \rangle\}$ is a transition from the current $\langle \text{state} \rangle$ to $\langle \text{state} \rangle + \langle \text{shift} \rangle$, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `\__regex_action_free:n` $\{\langle \text{shift} \rangle\}$, and `\__regex_action_free_group:n` $\{\langle \text{shift} \rangle\}$ are free transitions, which immediately perform the actions for the state $\langle \text{state} \rangle + \langle \text{shift} \rangle$ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `\__regex_action_submatch:nN` $\{\langle \text{group} \rangle\}$ $\langle \text{key} \rangle$ where the $\langle \text{key} \rangle$ is `<` or `>` for the beginning or end of group numbered $\langle \text{group} \rangle$. This causes the current position in the query to be stored as the $\langle \text{key} \rangle$ submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labeled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

<code>__regex_build:n</code> <code>__regex_build_aux:Nn</code> <code>__regex_build:N</code> <code>__regex_build_aux:NN</code>	The <code>n</code>-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of <code>capturing_group</code>). Finally, if the match reaches the last state, it is successful. A <code>false</code> boolean for argument <code>#1</code> for the auxiliaries will suppress the wildcard and make the match anchored: used for <code>\peek_regex:nTF</code> and similar.

```

6226 \cs_new_protected:Npn \__regex_build:n
6227   { \__regex_build_aux:Nn \c_true_bool }
6228 \cs_new_protected:Npn \__regex_build:N
6229   { \__regex_build_aux:NN \c_true_bool }
6230 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
6231   {
6232     \__regex_compile:n {#2}

```

```

6233     \__regex_build_aux:Nn #1 \l__regex_tmp_regex
6234   }
6235 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
6236   {
6237     \__regex_standard_escapechar:
6238     \int_zero:N \l__regex_capturing_group_int
6239     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6240     \__regex_build_new_state:
6241     \__regex_build_new_state:
6242     \__regex_toks_put_right:Nn \l__regex_left_state_int
6243     { \__regex_action_start_wildcard:N #1 }
6244     \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
6245     \__regex_toks_put_right:Nn \l__regex_right_state_int
6246     { \__regex_action_success: }
6247   }

```

(End of definition for __regex_build:n and others.)

`\g__regex_case_int` Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```

6248 \int_new:N \g__regex_case_int

```

(End of definition for \g__regex_case_int.)

`\l__regex_case_max_group_int` The largest group number appearing in any of the `<regex>` in the argument of `\regex-match_case:nn` and related functions.

```

6249 \int_new:N \l__regex_case_max_group_int

```

(End of definition for \l__regex_case_max_group_int.)

```

\__regex_case_build:n See \__regex_build:n, but with a loop.
\__regex_case_build:e
\__regex_case_build_aux:Nn
\__regex_case_build_loop:n
6250 \cs_new_protected:Npn \__regex_case_build:n #1
6251   {
6252     \__regex_case_build_aux:Nn \c_true_bool {#1}
6253     \int_gzero:N \g__regex_case_int
6254   }
6255 \cs_generate_variant:Nn \__regex_case_build:n { e }
6256 \cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
6257   {
6258     \__regex_standard_escapechar:
6259     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
6260     \__regex_build_new_state:
6261     \__regex_build_new_state:
6262     \__regex_toks_put_right:Nn \l__regex_left_state_int
6263     { \__regex_action_start_wildcard:N #1 }
6264     %
6265     \__regex_build_new_state:
6266     \__regex_toks_put_left:Ne \l__regex_left_state_int
6267     { \__regex_action_submatch:nN \c_zero_int < }
6268     \__regex_push_lr_states:
6269     \int_zero:N \l__regex_case_max_group_int
6270     \int_gzero:N \g__regex_case_int
6271     \tl_map_inline:nn {#2}
6272     {
6273       \int_gincr:N \g__regex_case_int

```

```

6274         \_regex_case_build_loop:n {##1}
6275     }
6276     \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
6277     \_regex_pop_lr_states:
6278 }
6279 \cs_new_protected:Npn \_regex_case_build_loop:n #1
6280 {
6281     \int_set_eq:NN \l__regex_capturing_group_int \c_one_int
6282     \_regex_compile_use:n {#1}
6283     \int_set:Nn \l__regex_case_max_group_int
6284     { \int_max:nn \l__regex_case_max_group_int \l__regex_capturing_group_int }
6285     \seq_pop:Nn \l__regex_right_state_seq \l__regex_tmpa_tl
6286     \int_set:Nn \l__regex_right_state_int \l__regex_tmpa_tl
6287     \_regex_toks_put_left:Ne \l__regex_right_state_int
6288     {
6289         \_regex_action_submatch:nN \c_zero_int >
6290         \int_gset:Nn \g__regex_case_int
6291         { \int_use:N \g__regex_case_int }
6292         \_regex_action_success:
6293     }
6294     \_regex_toks_clear:N \l__regex_max_state_int
6295     \seq_push:No \l__regex_right_state_seq
6296     { \int_use:N \l__regex_max_state_int }
6297     \int_incr:N \l__regex_max_state_int
6298 }

```

(End of definition for `\_regex_case_build:n`, `\_regex_case_build_aux:Nn`, and `\_regex_case_build_loop:n`.)

`\_regex_build_for_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g__regex_state_active_intarray` from `\l__regex_min_state_int` to `\l__regex_max_state_int`;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l__regex_min_state_int` to (the old) `\l__regex_max_state_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

6299 \cs_new_protected:Npn \_regex_build_for_cs:n #1
6300 {
6301     \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
6302     \_regex_build_new_state:
6303     \_regex_build_new_state:
6304     \_regex_push_lr_states:
6305     #1
6306     \_regex_pop_lr_states:
6307     \_regex_toks_put_right:Nn \l__regex_right_state_int
6308     {
6309         \if_int_compare:w -2 = \l__regex_curr_char_int

```



```

6310         \exp_after:wN \_regex_action_success:
6311         \fi:
6312     }
6313 }

```

(End of definition for _regex_build_for_cs:n.)

50.4.3 Helpers for building an nfa

_regex_push_lr_states: When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from T_EX's grouping.

```

6314 \cs_new_protected:Npn \_regex_push_lr_states:
6315 {
6316     \seq_push:Nn \l__regex_left_state_seq
6317     { \int_use:N \l__regex_left_state_int }
6318     \seq_push:Nn \l__regex_right_state_seq
6319     { \int_use:N \l__regex_right_state_int }
6320 }
6321 \cs_new_protected:Npn \_regex_pop_lr_states:
6322 {
6323     \seq_pop:Nn \l__regex_left_state_seq \l__regex_tmpa_tl
6324     \int_set:Nn \l__regex_left_state_int \l__regex_tmpa_tl
6325     \seq_pop:Nn \l__regex_right_state_seq \l__regex_tmpa_tl
6326     \int_set:Nn \l__regex_right_state_int \l__regex_tmpa_tl
6327 }

```

(End of definition for _regex_push_lr_states: and _regex_pop_lr_states:.)

_regex_build_transition_left:NNN
_regex_build_transition_right:nNn

Add a transition from #2 to #3 using the function #1. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```

6328 \cs_new_protected:Npn \_regex_build_transition_left:NNN #1#2#3
6329 { \_regex_toks_put_left:Ne #2 { #1 { \tex_the:D \_regex_int_eval:w #3 - #2 } } }
6330 \cs_new_protected:Npn \_regex_build_transition_right:nNn #1#2#3
6331 { \_regex_toks_put_right:Ne #2 { #1 { \tex_the:D \_regex_int_eval:w #3 - #2 } } }

```

(End of definition for _regex_build_transition_left:NNN and _regex_build_transition_right:nNn.)

_regex_build_new_state:

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```

6332 \cs_new_protected:Npn \_regex_build_new_state:
6333 {
6334     \_regex_toks_clear:N \l__regex_max_state_int
6335     \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
6336     \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
6337     \int_incr:N \l__regex_max_state_int
6338 }

```

(End of definition for _regex_build_new_state:.)

`\_regex_build_transitions_laziness:NNNN` This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by `#1`, true for lazy quantifiers, and false for greedy quantifiers.

```

6339 \cs_new_protected:Npn \_regex_build_transitions_laziness:NNNN #1#2#3#4#5
6340 {
6341   \_regex_build_new_state:
6342   \_regex_toks_put_right:Ne \l__regex_left_state_int
6343   {
6344     \if_meaning:w \c_true_bool #1
6345       #2 { \tex_the:D \_regex_int_eval:w #3 - \l__regex_left_state_int }
6346       #4 { \tex_the:D \_regex_int_eval:w #5 - \l__regex_left_state_int }
6347     \else:
6348       #4 { \tex_the:D \_regex_int_eval:w #5 - \l__regex_left_state_int }
6349       #2 { \tex_the:D \_regex_int_eval:w #3 - \l__regex_left_state_int }
6350     \fi:
6351   }
6352 }

```

(End of definition for `\_regex_build_transitions_laziness:NNNN`.)

50.4.4 Building classes

`\_regex_class:NnnnN` The arguments are: `<boolean> {<tests>} {<min>} {<more>} <laziness>`. First store the tests with a trailing `\_regex_action_cost:n`, in the true branch of `\_regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer `<more>` is 0 for fixed repetitions, `-1` for unbounded repetitions, and `<max> - <min>` for a range of repetitions.

```

6353 \cs_new_protected:Npn \_regex_class:NnnnN #1#2#3#4#5
6354 {
6355   \cs_set:Npe \_regex_tests_action_cost:n ##1
6356   {
6357     \exp_not:n { \exp_not:n {#2} }
6358     \bool_if:NTF #1
6359       { \_regex_break_point:TF { \_regex_action_cost:n {##1} } { } }
6360       { \_regex_break_point:TF { } { \_regex_action_cost:n {##1} } }
6361   }
6362   \if_case:w - #4 \exp_stop_f:
6363     \_regex_class_repeat:n {#3}
6364   \or: \_regex_class_repeat:nN {#3} #5
6365   \else: \_regex_class_repeat:nnN {#3} {#4} #5
6366   \fi:
6367 }
6368 \cs_new:Npn \_regex_tests_action_cost:n { \_regex_action_cost:n }

```

(End of definition for `\_regex_class:NnnnN` and `\_regex_tests_action_cost:n`.)

`\_regex_class_repeat:n` This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for `#1 = 0` repetitions: nothing is built.

```

6369 \cs_new_protected:Npn \_regex_class_repeat:n #1
6370 {
6371   \prg_replicate:nn {#1}
6372   {

```

```

6373     \__regex_build_new_state:
6374     \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
6375     \l__regex_left_state_int \l__regex_right_state_int
6376   }
6377 }

```

(End of definition for __regex_class_repeat:n.)

__regex_class_repeat:nN This implements unbounded repetitions of a single class (*e.g.* the * and + quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call __regex_class_repeat:n for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the laziness boolean #2.

```

6378 \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
6379 {
6380   \if_int_compare:w #1 = \c_zero_int
6381     \__regex_build_transitions_laziness:NNNN #2
6382     \__regex_action_free:n \l__regex_right_state_int
6383     \__regex_tests_action_cost:n \l__regex_left_state_int
6384   \else:
6385     \__regex_class_repeat:n {#1}
6386     \int_set_eq:NN \l__regex_tmpa_int \l__regex_left_state_int
6387     \__regex_build_transitions_laziness:NNNN #2
6388     \__regex_action_free:n \l__regex_right_state_int
6389     \__regex_action_free:n \l__regex_tmpa_int
6390   \fi:
6391 }

```

(End of definition for __regex_class_repeat:nN.)

__regex_class_repeat:nnN We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from max_state.

```

6392 \cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
6393 {
6394   \__regex_class_repeat:n {#1}
6395   \int_set:Nn \l__regex_tmpa_int
6396   { \l__regex_max_state_int + #2 - \c_one_int }
6397   \prg_replicate:nn { #2 }
6398   {
6399     \__regex_build_transitions_laziness:NNNN #3
6400     \__regex_action_free:n \l__regex_tmpa_int
6401     \__regex_tests_action_cost:n \l__regex_right_state_int
6402   }
6403 }

```

(End of definition for __regex_class_repeat:nnN.)

50.4.5 Building groups

`\__regex_group_aux:nnnnN` Arguments: $\{\langle label \rangle\} \{\langle contents \rangle\} \{\langle min \rangle\} \{\langle more \rangle\} \langle laziness \rangle$. If $\langle min \rangle$ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents `#2` of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly `#3` times, or `#3` or more times, or between `#3` and `#3 + #4` times, with laziness `#5`. The $\langle label \rangle$ `#1` is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```

6404 \cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
6405 {
6406     \if_int_compare:w #3 = \c_zero_int
6407         \__regex_build_new_state:
6408         \__regex_build_transition_right:nNn \__regex_action_free_group:n
6409         \l__regex_left_state_int \l__regex_right_state_int
6410     \fi:
6411     \__regex_build_new_state:
6412     \__regex_push_lr_states:
6413     #2
6414     \__regex_pop_lr_states:
6415     \if_case:w - #4 \exp_stop_f:
6416         \__regex_group_repeat:nn {#1} {#3}
6417     \or: \__regex_group_repeat:nnN {#1} {#3} #5
6418     \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
6419     \fi:
6420 }
```

(End of definition for `\__regex_group_aux:nnnnN`.)

`\__regex_group:nnnN` Hand to `\__regex_group_aux:nnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```

\__regex_group_no_capture:nnnN
6421 \cs_new_protected:Npn \__regex_group:nnnN #1
6422 {
6423     \exp_args:No \__regex_group_aux:nnnnN
6424     { \int_use:N \l__regex_capturing_group_int }
6425     {
6426         \int_incr:N \l__regex_capturing_group_int
6427         #1
6428     }
6429 }
6430 \cs_new_protected:Npn \__regex_group_no_capture:nnnN
6431 { \__regex_group_aux:nnnnN { -1 } }
```

(End of definition for `\__regex_group:nnnN` and `\__regex_group_no_capture:nnnN`.)

`\__regex_group_resetting:nnnN` Again, hand the label `-1` to `\__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `\__regex_branch:n \{\langle branch \rangle\}`.

```

6432 \cs_new_protected:Npn \__regex_group_resetting:nnnN #1
```

```

6433 {
6434   \__regex_group_aux:nnnnN { -1 }
6435   {
6436     \exp_args:Noo \__regex_group_resetting_loop:nnNn
6437     { \int_use:N \l__regex_capturing_group_int }
6438     { \int_use:N \l__regex_capturing_group_int }
6439     #1
6440     { ?? \prg_break:n } { }
6441     \prg_break_point:
6442   }
6443 }
6444 \cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
6445 {
6446   \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
6447   \int_set:Nn \l__regex_capturing_group_int {#2}
6448   #3 {#4}
6449   \exp_args:Ne \__regex_group_resetting_loop:nnNn
6450   { \int_max:nn {#1} \l__regex_capturing_group_int }
6451   {#2}
6452 }

```

(End of definition for __regex_group_resetting:nnnN and __regex_group_resetting_loop:nnNn.)

__regex_branch:n Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

6453 \cs_new_protected:Npn \__regex_branch:n #1
6454 {
6455   \__regex_build_new_state:
6456   \seq_get:NN \l__regex_left_state_seq \l__regex_tmpa_tl
6457   \int_set:Nn \l__regex_left_state_int \l__regex_tmpa_tl
6458   \__regex_build_transition_right:nnNn \__regex_action_free:n
6459   \l__regex_left_state_int \l__regex_right_state_int
6460   #1
6461   \seq_get:NN \l__regex_right_state_seq \l__regex_tmpa_tl
6462   \__regex_build_transition_right:nnNn \__regex_action_free:n
6463   \l__regex_right_state_int \l__regex_tmpa_tl
6464 }

```

(End of definition for __regex_branch:n.)

__regex_group_repeat:nn This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the capturing_group label). Otherwise, the auxiliary __regex_group_repeat_aux:n copies #2 times the \toks for the group, and leaves internal_a pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

6465 \cs_new_protected:Npn \__regex_group_repeat:nn #1#2
6466 {
6467   \if_int_compare:w #2 = \c_zero_int
6468     \int_set:Nn \l__regex_max_state_int
6469     { \l__regex_left_state_int - \c_one_int }
6470     \__regex_build_new_state:

```

```

6471     \else:
6472         \__regex_group_repeat_aux:n {#2}
6473         \__regex_group_submatches:nNN {#1}
6474         \l__regex_tmpa_int \l__regex_right_state_int
6475         \__regex_build_new_state:
6476     \fi:
6477 }

```

(End of definition for __regex_group_repeat:nn.)

__regex_group_submatches:nNN This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

6478 \cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
6479 {
6480     \if_int_compare:w #1 > - \c_one_int
6481         \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:n {#1} < }
6482         \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:n {#1} > }
6483     \fi:
6484 }

```

(End of definition for __regex_group_submatches:nNN.)

__regex_group_repeat_aux:n Here we repeat \toks ranging from left_state to max_state, #1 > 0 times. First add a transition so that the copies “chain” properly. Compute the shift c between the original copy and the last copy we want. Shift the right_state and max_state to their final values. We then want to perform c copy operations. At the end, b is equal to the max_state, and a points to the left of the last copy of the group.

```

6485 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
6486 {
6487     \__regex_build_transition_right:nNn \__regex_action_free:n
6488     \l__regex_right_state_int \l__regex_max_state_int
6489     \int_set_eq:NN \l__regex_tmpa_int \l__regex_left_state_int
6490     \int_set_eq:NN \l__regex_tmpb_int \l__regex_max_state_int
6491     \if_int_compare:w \__regex_int_eval:w #1 > \c_one_int
6492         \int_set:Nn \l__regex_tmpc_int
6493         {
6494             ( #1 - \c_one_int )
6495             * ( \l__regex_tmpb_int - \l__regex_tmpa_int )
6496         }
6497     \int_add:Nn \l__regex_right_state_int \l__regex_tmpc_int
6498     \int_add:Nn \l__regex_max_state_int \l__regex_tmpc_int
6499     \__regex_toks_memcpy:Nn
6500     \l__regex_tmpb_int
6501     \l__regex_tmpa_int
6502     \l__regex_tmpc_int
6503     \fi:
6504 }

```

(End of definition for __regex_group_repeat_aux:n.)

__regex_group_repeat:nnN This function is called to repeat a group at least n times; the case n = 0 is very different from n > 0. Assume first that n = 0. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state a

(remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `__regex_group_repeat_aux:n`.

```

6505 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
6506 {
6507   \if_int_compare:w #2 = \c_zero_int
6508     \__regex_group_submatches:nnN {#1}
6509     \l__regex_left_state_int \l__regex_right_state_int
6510     \int_set:Nn \l__regex_tmpa_int
6511     { \l__regex_left_state_int - \c_one_int }
6512     \__regex_build_transition_right:nNn \__regex_action_free:n
6513     \l__regex_right_state_int \l__regex_tmpa_int
6514     \__regex_build_new_state:
6515     \if_meaning:w \c_true_bool #3
6516       \__regex_build_transition_left:NNN \__regex_action_free:n
6517       \l__regex_tmpa_int \l__regex_right_state_int
6518     \else:
6519       \__regex_build_transition_right:nNn \__regex_action_free:n
6520       \l__regex_tmpa_int \l__regex_right_state_int
6521     \fi:
6522   \else:
6523     \__regex_group_repeat_aux:n {#2}
6524     \__regex_group_submatches:nnN {#1}
6525     \l__regex_tmpa_int \l__regex_right_state_int
6526     \if_meaning:w \c_true_bool #3
6527       \__regex_build_transition_right:nNn \__regex_action_free_group:n
6528       \l__regex_right_state_int \l__regex_tmpa_int
6529     \else:
6530       \__regex_build_transition_left:NNN \__regex_action_free_group:n
6531       \l__regex_right_state_int \l__regex_tmpa_int
6532     \fi:
6533     \__regex_build_new_state:
6534   \fi:
6535 }

```

(End of definition for `__regex_group_repeat:nnN`.)

`__regex_group_repeat:nnnN`

We wish to repeat the group between `#2` and `#2 + #3` times, with a laziness controlled by `#4`. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first `#2` copies of the group, but that forces us to treat specially the case `#2 = 0`. Repeat that group with submatch tracking `#2 + #3` times (the maximum number of repetitions). Then our goal is to add `#3` transitions from the end of the `#2`-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with `#2 = 0`, the transition which skips over all

copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

6536 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
6537 {
6538   \__regex_group_submatches:nnN {#1}
6539   \l__regex_left_state_int \l__regex_right_state_int
6540   \__regex_group_repeat_aux:n { #2 + #3 }
6541   \if_meaning:w \c_true_bool #4
6542     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
6543     \prg_replicate:nn { #3 }
6544     {
6545       \int_sub:Nn \l__regex_left_state_int
6546       { \l__regex_tmpb_int - \l__regex_tmpa_int }
6547       \__regex_build_transition_left:NNN \__regex_action_free:n
6548       \l__regex_left_state_int \l__regex_max_state_int
6549     }
6550   \else:
6551     \prg_replicate:nn { #3 - \c_one_int }
6552     {
6553       \int_sub:Nn \l__regex_right_state_int
6554       { \l__regex_tmpb_int - \l__regex_tmpa_int }
6555       \__regex_build_transition_right:nNn \__regex_action_free:n
6556       \l__regex_right_state_int \l__regex_max_state_int
6557     }
6558     \if_int_compare:w #2 = \c_zero_int
6559       \int_set:Nn \l__regex_right_state_int
6560       { \l__regex_left_state_int - \c_one_int }
6561     \else:
6562       \int_sub:Nn \l__regex_right_state_int
6563       { \l__regex_tmpb_int - \l__regex_tmpa_int }
6564     \fi:
6565     \__regex_build_transition_right:nNn \__regex_action_free:n
6566     \l__regex_right_state_int \l__regex_max_state_int
6567   \fi:
6568   \__regex_build_new_state:
6569 }

```

(End of definition for __regex_group_repeat:nnnN.)

50.4.6 Others

__regex_assertion:Nn Usage: __regex_assertion:Nn <boolean> {<test>}, where the <test> is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The __regex_A_test: test is used by the \A escape: check if the last boundary-markers of the string are non-word characters for this purpose. The __regex_G_test: test is used by the \G escape: check if the current character is a word character or not, and do the same to the last character. The __regex_Z_test: test is used by the \Z escape: check if the current character is a non-word character or not, and do the same to the last character.

```

6570 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
6571 {
6572   \__regex_build_new_state:
6573   \__regex_toks_put_right:Ne \l__regex_left_state_int
6574   {
6575     \exp_not:n {#2}
6576     \__regex_break_point:TF

```



```

6577         \bool_if:NF #1 { { } }
6578     {
6579         \__regex_action_free:n
6580         {
6581             \tex_the:D \__regex_int_eval:w
6582             \l__regex_right_state_int - \l__regex_left_state_int
6583         }
6584     }
6585     \bool_if:NT #1 { { } }
6586 }
6587 }
6588 \cs_new_protected:Npn \__regex_b_test:
6589 {
6590     \group_begin:
6591     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
6592     \__regex_prop_w:
6593     \__regex_break_point:TF
6594     { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
6595     { \group_end: \__regex_prop_w: }
6596 }
6597 \cs_new_protected:Npn \__regex_Z_test:
6598 {
6599     \if_int_compare:w -2 = \l__regex_curr_char_int
6600     \exp_after:wN \__regex_break_true:w
6601     \fi:
6602 }
6603 \cs_new_protected:Npn \__regex_A_test:
6604 {
6605     \if_int_compare:w -2 = \l__regex_last_char_int
6606     \exp_after:wN \__regex_break_true:w
6607     \fi:
6608 }
6609 \cs_new_protected:Npn \__regex_G_test:
6610 {
6611     \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
6612     \exp_after:wN \__regex_break_true:w
6613     \fi:
6614 }

```

(End of definition for __regex_assertion:Nn and others.)

__regex_command_K: Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

6615 \cs_new_protected:Npn \__regex_command_K:
6616 {
6617     \__regex_build_new_state:
6618     \__regex_toks_put_right:Ne \l__regex_left_state_int
6619     {
6620         \__regex_action_submatch:nN \c_zero_int <
6621         \bool_set_true:N \l__regex_fresh_thread_bool
6622         \__regex_action_free:n
6623         {
6624             \tex_the:D \__regex_int_eval:w
6625             \l__regex_right_state_int - \l__regex_left_state_int

```

```

6626         }
6627         \bool_set_false:N \l__regex_fresh_thread_bool
6628     }
6629 }

```

(End of definition for `\__regex_command_K:`.)

50.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_info_intarray` (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

50.5.1 Variables used when matching

```

\l__regex_min_pos_int
\l__regex_max_pos_int
\l__regex_curr_pos_int
\l__regex_start_pos_int
\l__regex_success_pos_int

```

The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```

6630 \int_new:N \l__regex_min_pos_int
6631 \int_new:N \l__regex_max_pos_int
6632 \int_new:N \l__regex_curr_pos_int
6633 \int_new:N \l__regex_start_pos_int
6634 \int_new:N \l__regex_success_pos_int

```

(End of definition for `\l__regex_min_pos_int` and others.)

`\l__regex_curr_char_int` The character and category codes of the token at the current position and a token list
`\l__regex_curr_catcode_int` expanding to that token; the character code of the token at the previous position; the
`\l__regex_curr_token_tl` character code of the token just before a successful match; and the character code of the
`\l__regex_last_char_int` result of changing the case of the current token (A-Z↔a-z). This last integer is only
`\l__regex_last_char_success_int` computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also
`\l__regex_case_changed_char_int` used in various other phases to hold a character code.

```
6635 \int_new:N \l__regex_curr_char_int
6636 \int_new:N \l__regex_curr_catcode_int
6637 \tl_new:N \l__regex_curr_token_tl
6638 \int_new:N \l__regex_last_char_int
6639 \int_new:N \l__regex_last_char_success_int
6640 \int_new:N \l__regex_case_changed_char_int
```

(End of definition for `\l__regex_curr_char_int` and others.)

`\l__regex_curr_state_int` For every character in the token list, each of the active states is considered in turn.
The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently
considered: transitions are then given as shifts relative to the current state.

```
6641 \int_new:N \l__regex_curr_state_int
```

(End of definition for `\l__regex_curr_state_int`.)

`\l__regex_curr_submatches_tl` The submatches for the thread which is currently active are stored in the `curr_-`
`\l__regex_success_submatches_tl` `submatches` list, which is almost a comma list, but ends with a comma. This list is stored
by `\__regex_store_state:n` into an intarray variable, to be retrieved when matching at
the next position. When a thread succeeds, this list is copied to `\l__regex_success_-`
`submatches_tl`: only the last successful thread remains there.

```
6642 \tl_new:N \l__regex_curr_submatches_tl
6643 \tl_new:N \l__regex_success_submatches_tl
```

(End of definition for `\l__regex_curr_submatches_tl` and `\l__regex_success_submatches_tl`.)

`\l__regex_step_int` This integer, always even, is increased every time a character in the query is read, and not
reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the
last step in which each `<state>` in the NFA was encountered. This lets us break infinite
loops by not visiting the same state twice in the same step. In fact, the step we store
is equal to `step` when we have started performing the operations of `\toks<state>`, but
not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_-`
`active_intarray`. This is needed to track submatches properly (see building phase).
The `step` is also used to attach each set of submatch information to a given iteration
(and automatically discard it when it corresponds to a past step).

```
6644 \int_new:N \l__regex_step_int
```

(End of definition for `\l__regex_step_int`.)

`\l__regex_min_thread_int` All the currently active threads are kept in order of precedence in `\g__regex_thread_-`
`\l__regex_max_thread_int` `info_intarray` together with the corresponding submatch information. Data in this
intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded).
At the start of every step, the whole array is unpacked, so that the space can immediately
be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
6645 \int_new:N \l__regex_min_thread_int
6646 \int_new:N \l__regex_max_thread_int
```

(End of definition for `\l__regex_min_thread_int` and `\l__regex_max_thread_int`.)

`\g__regex_state_active_intarray` `\g__regex_state_active_intarray` stores the last *<step>* in which each *<state>* was active. `\g__regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
6647 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
```

```
6648 \intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(End of definition for `\g__regex_state_active_intarray` and `\g__regex_thread_info_intarray`.)

`\l__regex_matched_analysis_tl` `\l__regex_curr_analysis_tl` The list `\l__regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l__regex_matched_analysis_tl` (constructed under the `tl_build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
6649 \tl_new:N \l__regex_matched_analysis_tl
```

```
6650 \tl_new:N \l__regex_curr_analysis_tl
```

(End of definition for `\l__regex_matched_analysis_tl` and `\l__regex_curr_analysis_tl`.)

`\l__regex_every_match_tl` Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `\__regex_single_match:` and `\__regex_multi_match:n`.

```
6651 \tl_new:N \l__regex_every_match_tl
```

(End of definition for `\l__regex_every_match_tl`.)

`\l__regex_fresh_thread_bool` `\l__regex_empty_success_bool` `\__regex_if_two_empty_matches:F` When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `\__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
6652 \bool_new:N \l__regex_fresh_thread_bool
```

```
6653 \bool_new:N \l__regex_empty_success_bool
```

```
6654 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(End of definition for `\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, and `\__regex_if_two_empty_matches:F`.)

`\g__regex_success_bool` `\l__regex_saved_success_bool` `\l__regex_match_success_bool` The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
6655 \bool_new:N \g__regex_success_bool
```

```
6656 \bool_new:N \l__regex_saved_success_bool
```

```
6657 \bool_new:N \l__regex_match_success_bool
```

(End of definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex-match_success_bool`.)

50.5.2 Matching: framework

```

__regex_match:n Initialize the variables that should be set once for each user function (even for multiple
__regex_match_cs:n matches). Namely, the overall matching is not yet successful; none of the states should
__regex_match_init: be marked as visited (\g__regex_state_active_intarray), and we start at step 0; we
pretend that there was a previous match ending at the start of the query, which was not
empty (to avoid smothering an empty match at the start). Once all this is set up, we are
ready for the ride. Find the first match.

6658 \cs_new_protected:Npn \__regex_match:n #1
6659 {
6660   \__regex_match_init:
6661   \__regex_match_once_init:
6662   \tl_analysis_map_inline:nn {#1}
6663   { \__regex_match_one_token:nnN {##1} {##2} ##3 }
6664   \__regex_match_one_token:nnN { } { -2 } F
6665   \prg_break_point:Nn \__regex_maplike_break: { }
6666 }
6667 \cs_new_protected:Npn \__regex_match_cs:n #1
6668 {
6669   \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
6670   \__regex_match_init:
6671   \__regex_match_once_init:
6672   \str_map_inline:nn {#1}
6673   {
6674     \tl_if_blank:nTF {##1}
6675     { \__regex_match_one_token:nnN {##1} {'##1} A }
6676     { \__regex_match_one_token:nnN {##1} {'##1} C }
6677   }
6678   \__regex_match_one_token:nnN { } { -2 } F
6679   \prg_break_point:Nn \__regex_maplike_break: { }
6680 }
6681 \cs_new_protected:Npn \__regex_match_init:
6682 {
6683   \bool_gset_false:N \g__regex_success_bool
6684   \int_step_inline:nnn
6685   \l__regex_min_state_int { \l__regex_max_state_int - \c_one_int }
6686   {
6687     \__kernel_intarray_gset:Nnn
6688     \g__regex_state_active_intarray {##1} \c_one_int
6689   }
6690   \int_zero:N \l__regex_step_int
6691   \int_set:Nn \l__regex_min_pos_int { 2 }
6692   \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
6693   \int_set:Nn \l__regex_last_char_success_int { -2 }
6694   \tl_build_begin:N \l__regex_matched_analysis_tl
6695   \tl_clear:N \l__regex_curr_analysis_tl
6696   \int_set_eq:NN \l__regex_min_submatch_int \c_one_int
6697   \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
6698   \bool_set_false:N \l__regex_empty_success_bool
6699 }

```

(End of definition for `\__regex_match:n`, `\__regex_match_cs:n`, and `\__regex_match_init:.`)

`\__regex_match_once_init:` This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `\__regex_match_one_token:nnN` increments `\l__regex_curr_pos_int` and saves `\l__regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```

6700 \cs_new_protected:Npn \__regex_match_once_init:
6701 {
6702   \if_meaning:w \c_true_bool \l__regex_empty_success_bool
6703     \cs_set:Npn \__regex_if_two_empty_matches:F
6704     {
6705       \int_compare:nNnF
6706         \l__regex_start_pos_int = \l__regex_curr_pos_int
6707     }
6708   \else:
6709     \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
6710   \fi:
6711   \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
6712   \bool_set_false:N \l__regex_match_success_bool
6713   \tl_set:Ne \l__regex_curr_submatches_tl
6714     { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
6715   \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6716   \__regex_store_state:n { \l__regex_min_state_int }
6717   \int_set:Nn \l__regex_curr_pos_int { \l__regex_start_pos_int - \c_one_int }
6718   \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
6719   \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_tmpa_tl
6720   \exp_args:NNf \__regex_match_once_init_aux:
6721   \tl_map_inline:nn
6722     { \exp_after:wN \l__regex_tmpa_tl \l__regex_curr_analysis_tl }
6723     { \__regex_match_one_token:nnN #1 }
6724   \prg_break_point:Nn \__regex_maplike_break: { }
6725 }
6726 \cs_new_protected:Npn \__regex_match_once_init_aux:
6727 {
6728   \tl_build_begin:N \l__regex_matched_analysis_tl
6729   \tl_clear:N \l__regex_curr_analysis_tl
6730 }

```

(End of definition for `\__regex_match_once_init:.`)

`\__regex_single_match:` For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```

6731 \cs_new_protected:Npn \__regex_single_match:
6732 {
6733   \tl_set:Nn \l__regex_every_match_tl
6734   {

```

```

6735         \bool_gset_eq:NN
6736         \g__regex_success_bool
6737         \l__regex_match_success_bool
6738         \__regex_maplike_break:
6739     }
6740 }
6741 \cs_new_protected:Npn \__regex_multi_match:n #1
6742 {
6743     \tl_set:Nn \l__regex_every_match_tl
6744     {
6745         \if_meaning:w \c_false_bool \l__regex_match_success_bool
6746         \exp_after:wN \__regex_maplike_break:
6747         \fi:
6748         \bool_gset_true:N \g__regex_success_bool
6749         #1
6750         \__regex_match_once_init:
6751     }
6752 }

```

(End of definition for __regex_single_match: and __regex_multi_match:n.)

__regex_match_one_token:nnN __regex_match_one_active:n

At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_thread`). This results in a sequence of `\__regex_use_state_and_submatches:w` `<state>`, `<submatch-clist>` `\__regex_sep:` and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `\__regex_action_wildcard:`.

```

6753 \cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
6754 {
6755     \int_add:Nn \l__regex_step_int { 2 }
6756     \int_incr:N \l__regex_curr_pos_int
6757     \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
6758     \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
6759     \tl_set:Nn \l__regex_curr_token_tl {#1}
6760     \int_set:Nn \l__regex_curr_char_int {#2}
6761     \int_set:Nn \l__regex_curr_catcode_int { "#3 }
6762     \tl_build_put_right:Ne \l__regex_matched_analysis_tl
6763     { \exp_not:o \l__regex_curr_analysis_tl }
6764     \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
6765     \use:e
6766     {
6767         \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6768         \int_step_function:nnN
6769         \l__regex_min_thread_int
6770         { \l__regex_max_thread_int - \c_one_int }
6771         \__regex_match_one_active:n
6772     }
6773     \prg_break_point:
6774     \bool_set_false:N \l__regex_fresh_thread_bool
6775     \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
6776     \if_int_compare:w -2 < \l__regex_curr_char_int
6777         \exp_after:wN \use_i:nn

```

```

6778     \fi:
6779 \fi:
6780 \l__regex_every_match_tl
6781 }
6782 \cs_new:Npn \__regex_match_one_active:n #1
6783 {
6784     \__regex_use_state_and_submatches:w
6785     \__kernel_intarray_range_to_clist:Nnn
6786     \g__regex_thread_info_intarray
6787     { \c_one_int + #1 * (\l__regex_capturing_group_int * 2 + \c_one_int) }
6788     { (\c_one_int + #1) * (\l__regex_capturing_group_int * 2 + \c_one_int) }
6789     \__regex_sep:
6790 }

```

(End of definition for __regex_match_one_token:nnN and __regex_match_one_active:n.)

50.5.3 Using states of the nfa

__regex_use_state: Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as **step + 1**: any thread hitting it at that point will be terminated.

```

6791 \cs_new_protected:Npn \__regex_use_state:
6792 {
6793     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6794     \l__regex_curr_state_int \l__regex_step_int
6795     \__regex_toks_use:w \l__regex_curr_state_int
6796     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6797     \l__regex_curr_state_int
6798     { \__regex_int_eval:w \l__regex_step_int + \c_one_int \scan_stop: }
6799 }

```

(End of definition for __regex_use_state:.)

__regex_use_state_and_submatches:w This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```

6800 \cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 \__regex_sep:
6801 {
6802     \int_set:Nn \l__regex_curr_state_int {#1}
6803     \if_int_compare:w
6804         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6805         \l__regex_curr_state_int
6806         < \l__regex_step_int
6807     \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
6808     \exp_after:wN \__regex_use_state:
6809     \fi:
6810     \scan_stop:
6811 }

```

(End of definition for __regex_use_state_and_submatches:w.)

50.5.4 Actions when matching

`\__regex_action_start_wildcard:N` For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `\__regex_match_one_token:nnN` too.

```

6812 \cs_new_protected:Npn \__regex_action_start_wildcard:N #1
6813 {
6814   \bool_set_true:N \l__regex_fresh_thread_bool
6815   \__regex_action_free:n {1}
6816   \bool_set_false:N \l__regex_fresh_thread_bool
6817   \bool_if:NT #1 { \__regex_action_cost:n {0} }
6818 }
```

(End of definition for `\__regex_action_start_wildcard:N`.)

`\__regex_action_free:n` These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```

6819 \cs_new_protected:Npn \__regex_action_free:n
6820 { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
6821 \cs_new_protected:Npn \__regex_action_free_group:n
6822 { \__regex_action_free_aux:nn { < \l__regex_step_int } }
6823 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
6824 {
6825   \use:e
6826   {
6827     \int_add:Nn \l__regex_curr_state_int {#2}
6828     \exp_not:n
6829     {
6830       \if_int_compare:w
6831         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6832         \l__regex_curr_state_int
6833         #1
6834         \exp_after:wN \__regex_use_state:
6835         \fi:
6836     }
6837     \int_set:Nn \l__regex_curr_state_int
6838     { \int_use:N \l__regex_curr_state_int }
6839     \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
6840     { \exp_not:o \l__regex_curr_submatches_tl }
6841   }
6842 }
```

(End of definition for `\__regex_action_free:n`, `\__regex_action_free_group:n`, and `\__regex_action_free_aux:nn`.)

`\__regex_action_cost:n` A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```

6843 \cs_new_protected:Npn \__regex_action_cost:n #1
6844 {
6845   \exp_args:No \__regex_store_state:n
6846   { \tex_the:D \__regex_int_eval:w \l__regex_curr_state_int + #1 }
6847 }

```

(End of definition for __regex_action_cost:n.)

__regex_store_state:n Put the given state and current submatch information in \g__regex_thread_info_intarray, and increment the length of the array.

```

6848 \cs_new_protected:Npn \__regex_store_state:n #1
6849 {
6850   \exp_args:No \__regex_store_submatches:nn
6851   \l__regex_curr_submatches_tl {#1}
6852   \int_incr:N \l__regex_max_thread_int
6853 }
6854 \cs_new_protected:Npn \__regex_store_submatches:nn #1#2
6855 {
6856   \__kernel_intarray_gset_range_from_clist:Nnn
6857   \g__regex_thread_info_intarray
6858   {
6859     \__regex_int_eval:w
6860     \c_one_int + \l__regex_max_thread_int *
6861     (\l__regex_capturing_group_int * 2 + \c_one_int)
6862   }
6863   { #2 , #1 }
6864 }

```

(End of definition for __regex_store_state:n and __regex_store_submatches:.)

__regex_disable_submatches: Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

6865 \cs_new_protected:Npn \__regex_disable_submatches:
6866 {
6867   \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
6868   \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
6869 }

```

(End of definition for __regex_disable_submatches:.)

__regex_action_submatch:nN Update the current submatches with the information from the current position. Maybe a bottleneck.

```

\__regex_action_submatch_aux:w
\__regex_action_submatch_auxii:w
\__regex_action_submatch_auxiii:w
\__regex_action_submatch_auxiv:w
6870 \cs_new_protected:Npn \__regex_action_submatch:nN #1#2
6871 {
6872   \exp_after:wN \__regex_action_submatch_aux:w
6873   \l__regex_curr_submatches_tl \__regex_sep: {#1} #2
6874 }
6875 \cs_new_protected:Npn \__regex_action_submatch_aux:w #1 \__regex_sep: #2#3
6876 {
6877   \tl_set:Nx \l__regex_curr_submatches_tl
6878   {
6879     \prg_replicate:nn
6880     { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }

```

```

6881         { \__regex_action_submatch_auxii:w }
6882         \__regex_action_submatch_auxiii:w
6883         #1
6884     }
6885 }
6886 \cs_new:Npn \__regex_action_submatch_auxii:w
6887     #1 \__regex_action_submatch_auxiii:w #2 ,
6888     { #2 , #1 \__regex_action_submatch_auxiii:w }
6889 \cs_new:Npn \__regex_action_submatch_auxiii:w #1 ,
6890     { \int_use:N \l__regex_curr_pos_int , }

```

(End of definition for __regex_action_submatch:nN and others.)

__regex_action_success: There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with \prg_break:, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

6891 \cs_new_protected:Npn \__regex_action_success:
6892 {
6893     \__regex_if_two_empty_matches:F
6894     {
6895         \bool_set_true:N \l__regex_match_success_bool
6896         \bool_set_eq:NN \l__regex_empty_success_bool
6897         \l__regex_fresh_thread_bool
6898         \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
6899         \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
6900         \tl_build_begin:N \l__regex_matched_analysis_tl
6901         \tl_set_eq:NN \l__regex_success_submatches_tl
6902         \l__regex_curr_submatches_tl
6903         \prg_break:
6904     }
6905 }

```

(End of definition for __regex_action_success:.)

50.6 Replacement

50.6.1 Variables and helpers used in replacement

\l__regex_replacement_csnames_int The behavior of closing braces inside a replacement text depends on whether a sequences \c{ or \u{ has been encountered. The number of “open” such sequences that should be closed by } is stored in \l__regex_replacement_csnames_int, and decreased by 1 by each }.

```

6906 \int_new:N \l__regex_replacement_csnames_int

```

(End of definition for \l__regex_replacement_csnames_int.)

\l__regex_replacement_category_tl This sequence of letters is used to correctly restore categories in nested constructions such as \cL(abc\cD(_)d).

\l__regex_replacement_category_seq

```

6907 \tl_new:N \l__regex_replacement_category_tl
6908 \seq_new:N \l__regex_replacement_category_seq

```

(End of definition for `\l__regex_replacement_category_tl` and `\l__regex_replacement_category_seq`.)

`\g__regex_balance_tl` This token list holds the replacement text for `\__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
6909 \tl_new:N \g__regex_balance_tl
```

(End of definition for `\g__regex_balance_tl`.)

`\__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
6910 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
6911 { - \__regex_submatch_balance:n {#1} }
```

(End of definition for `\__regex_replacement_balance_one_match:n`.)

`\__regex_replacement_do_one_match:n` The input is the same as `\__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
6912 \cs_new:Npn \__regex_replacement_do_one_match:n #1
6913 {
6914   \__regex_query_range:nn
6915   { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
6916   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6917 }
```

(End of definition for `\__regex_replacement_do_one_match:n`.)

`\__regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an e/x-expanding assignment, `\exp_not:N #` behaves as a single #, whereas `\exp_not:n {#}` behaves as a doubled ##.

```
6918 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End of definition for `\__regex_replacement_exp_not:N`.)

`\__regex_replacement_exp_not:V` This is used for the implementation of `\u`, and it gets redefined for `\peek_regex_replace_once:nnTF`.

```
6919 \cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(End of definition for `\__regex_replacement_exp_not:V`.)

50.6.2 Query and brace balance

`\_regex_query_range:nn` When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l\_regex_min_pos_int` inclusive to `\l\_regex_max_pos_int` exclusive. The function `\_regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max>-1` included. Once this is expanded, a second e-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```

6920 \cs_new:Npn \_regex_query_range:nn #1#2
6921 {
6922   \exp_after:wN \_regex_query_range_loop:ww
6923   \int_value:w \_regex_int_eval:w #1 \exp_after:wN \_regex_sep:
6924   \int_value:w \_regex_int_eval:w #2 \_regex_sep:
6925   \prg_break_point:
6926 }
6927 \cs_new:Npn \_regex_query_range_loop:ww #1 \_regex_sep: #2 \_regex_sep:
6928 {
6929   \if_int_compare:w #1 < #2 \exp_stop_f:
6930   \else:
6931     \prg_break:n
6932   \fi:
6933   \_regex_toks_use:w #1 \exp_stop_f:
6934   \exp_after:wN \_regex_query_range_loop:ww
6935   \int_value:w \_regex_int_eval:w #1 + \c_one_int \_regex_sep: #2 \_regex_sep:
6936 }

```

(End of definition for `\_regex_query_range:nn` and `\_regex_query_range_loop:ww`.)

`\_regex_query_submatch:n` Find the start and end positions for a given submatch (of a given match).

```

6937 \cs_new:Npn \_regex_query_submatch:n #1
6938 {
6939   \_regex_query_range:nn
6940   { \_kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
6941   { \_kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
6942 }

```

(End of definition for `\_regex_query_submatch:n`.)

`\_regex_submatch_balance:n` Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

6943 \cs_new_protected:Npn \_regex_submatch_balance:n #1
6944 {
6945   \tex_the:D \_regex_int_eval:w
6946   \_regex_intarray_item:NnF \g__regex_balance_intarray
6947   {
6948     \_kernel_intarray_item:Nn
6949     \g__regex_submatch_end_intarray {#1}
6950   }
6951   \c_zero_int
6952   -

```

```

6953     \__regex_intarray_item:NnF \g__regex_balance_intarray
6954     {
6955         \__kernel_intarray_item:Nn
6956         \g__regex_submatch_begin_intarray {#1}
6957     }
6958     \c_zero_int
6959 \scan_stop:
6960 }

```

(End of definition for __regex_submatch_balance:n.)

50.6.3 Framework

```

\__regex_replacement:n
\__regex_replacement:e
  \__regex_replacement_apply:Nn
\__regex_replacement_set:n

```

The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \g__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.

```

6961 \cs_new_protected:Npn \__regex_replacement:n
6962   { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
6963 \cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
6964   {
6965     \group_begin:
6966     \tl_build_begin:N \l__regex_build_tl
6967     \int_zero:N \l__regex_balance_int
6968     \tl_gclear:N \g__regex_balance_tl
6969     \__regex_escape_use:nnnn
6970     {
6971       \if_charcode:w \c_right_brace_str ##1
6972         \__regex_replacement_rbrace:N
6973       \else:
6974         \if_charcode:w \c_left_brace_str ##1
6975           \__regex_replacement_lbrace:N
6976         \else:
6977           \__regex_replacement_normal:n
6978         \fi:
6979       \fi:
6980       ##1
6981     }
6982     { \__regex_replacement_escaped:N ##1 }
6983     { \__regex_replacement_normal:n ##1 }
6984     {#2}
6985     \prg_do_nothing: \prg_do_nothing:
6986     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
6987       \msg_error:nne { regex } { replacement-missing-rbrace }
6988       { \int_use:N \l__regex_replacement_csnames_int }
6989       \tl_build_put_right:Ne \l__regex_build_tl
6990       { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
6991     \fi:
6992     \seq_if_empty:NF \l__regex_replacement_category_seq
6993     {

```

```

6994         \msg_error:nne { regex } { replacement-missing-rparen }
6995         { \seq_count:N \l__regex_replacement_category_seq }
6996         \seq_clear:N \l__regex_replacement_category_seq
6997     }
6998     \tl_gput_right:Nx \g__regex_balance_tl
6999     { + \int_use:N \l__regex_balance_int }
7000     \tl_build_end:N \l__regex_build_tl
7001     \exp_args:NNo
7002     \group_end:
7003     #1 \l__regex_build_tl
7004 }
7005 \cs_generate_variant:Nn \__regex_replacement:n { e }
7006 \cs_new_protected:Npn \__regex_replacement_set:n #1
7007 {
7008     \cs_set:Npn \__regex_replacement_do_one_match:n ##1
7009     {
7010         \__regex_query_range:nn
7011         {
7012             \__kernel_intarray_item:Nn
7013             \g__regex_submatch_prev_intarray {##1}
7014         }
7015         {
7016             \__kernel_intarray_item:Nn
7017             \g__regex_submatch_begin_intarray {##1}
7018         }
7019         #1
7020     }
7021     \exp_args:Nno \use:n
7022     { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
7023     {
7024         \g__regex_balance_tl
7025         - \__regex_submatch_balance:n {##1}
7026     }
7027 }

```

(End of definition for `\__regex_replacement:n`, `\__regex_replacement_apply:Nn`, and `\__regex_replacement_set:n`.)

```

\__regex_case_replacement:n
\__regex_case_replacement:e

```

```

7028 \tl_new:N \g__regex_case_replacement_tl
7029 \tl_new:N \g__regex_case_balance_tl
7030 \cs_new_protected:Npn \__regex_case_replacement:n #1
7031 {
7032     \tl_gset:Nn \g__regex_case_balance_tl
7033     {
7034         \if_case:w
7035         \__kernel_intarray_item:Nn
7036         \g__regex_submatch_case_intarray {##1}
7037     }
7038     \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
7039     \tl_map_tokens:nn {#1}
7040     { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
7041     \tl_gset:Nn \g__regex_balance_tl
7042     { \g__regex_case_balance_tl \fi: }

```

```

7043 \exp_args:No \__regex_replacement_set:n
7044 { \g__regex_case_replacement_tl \fi: }
7045 }
7046 \cs_generate_variant:Nn \__regex_case_replacement:n { e }
7047 \cs_new_protected:Npn \__regex_case_replacement_aux:n #1
7048 {
7049   \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
7050   \tl_gput_right:No \g__regex_case_balance_tl
7051   { \exp_after:wN \or: \g__regex_balance_tl }
7052 }

```

(End of definition for __regex_case_replacement:n.)

__regex_replacement_put:n This gets redefined for \peek_regex_replace_once:nnTF.

```

7053 \cs_new_protected:Npn \__regex_replacement_put:n
7054 { \tl_build_put_right:Nn \l__regex_build_tl }

```

(End of definition for __regex_replacement_put:n.)

__regex_replacement_normal:n
 __regex_replacement_normal_aux:N

Most characters are simply sent to the output by \tl_build_put_right:Nn, unless a particular category code has been requested: then __regex_replacement_c_A:w or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of \l__regex_replacement_category_tl. The argument #1 is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we take into account the current catcode régime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to “other”.

```

7055 \cs_new_protected:Npn \__regex_replacement_normal:n #1
7056 {
7057   \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
7058   { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
7059   {
7060     \tl_if_empty:NNTF \l__regex_replacement_category_tl
7061     { \__regex_replacement_normal_aux:N #1 }
7062     { % (
7063       \token_if_eq_charcode:NNTF #1 )
7064       {
7065         \seq_pop:NN \l__regex_replacement_category_seq
7066         \l__regex_replacement_category_tl
7067       }
7068       {
7069         \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
7070         ? #1
7071       }
7072     }
7073   }
7074 }
7075 \cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
7076 {
7077   \token_if_eq_charcode:NNTF #1 \c_space_token
7078   { \__regex_replacement_c_S:w }
7079   {
7080     \exp_after:wN \exp_after:wN

```



```

7081         \if_case:w \tex_catcode:D '#1 \exp_stop_f:
7082             \__regex_replacement_c_0:w
7083         \or: \__regex_replacement_c_B:w
7084         \or: \__regex_replacement_c_E:w
7085         \or: \__regex_replacement_c_M:w
7086         \or: \__regex_replacement_c_T:w
7087         \or: \__regex_replacement_c_0:w
7088         \or: \__regex_replacement_c_P:w
7089         \or: \__regex_replacement_c_U:w
7090         \or: \__regex_replacement_c_D:w
7091         \or: \__regex_replacement_c_0:w
7092         \or: \__regex_replacement_c_S:w
7093         \or: \__regex_replacement_c_L:w
7094         \or: \__regex_replacement_c_0:w
7095         \or: \__regex_replacement_c_A:w
7096         \else: \__regex_replacement_c_0:w
7097     \fi:
7098 }
7099 ? #1
7100 }

```

(End of definition for `\__regex_replacement_normal:n` and `\__regex_replacement_normal_aux:N`.)

`\__regex_replacement_escaped:N`

As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```

7101 \cs_new_protected:Npn \__regex_replacement_escaped:N #1
7102 {
7103     \cs_if_exist_use:cF { __regex_replacement_#1:w }
7104     {
7105         \if_int_compare:w \c_one_int < 1#1 \exp_stop_f:
7106         \__regex_replacement_put_submatch:n {#1}
7107     \else:
7108         \__regex_replacement_normal:n {#1}
7109     \fi:
7110 }
7111 }

```

(End of definition for `\__regex_replacement_escaped:N`.)

50.6.4 Submatches

`\__regex_replacement_put_submatch:n`
`\__regex_replacement_put_submatch_aux:n`

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

```

7112 \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
7113 {
7114     \if_int_compare:w #1 < \l__regex_capturing_group_int
7115     \__regex_replacement_put_submatch_aux:n {#1}
7116 \else:
7117     \msg_expandable_error:nnff { regex } { submatch-too-big }
7118     {#1} { \int_eval:n { \l__regex_capturing_group_int - \c_one_int } }
7119 \fi:

```

```

7120 }
7121 \cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
7122 {
7123   \tl_build_put_right:Nn \l__regex_build_tl
7124   { \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
7125   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7126   \tl_gput_right:Nn \g__regex_balance_tl
7127   { + \__regex_submatch_balance:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
7128   \fi:
7129 }

```

(End of definition for __regex_replacement_put_submatch:n and __regex_replacement_put_submatch_aux:n.)

__regex_replacement_g:w Grab digits for the \g escape sequence in a primitive assignment to the integer \l__-
 __regex_replacement_g_digits:NN regex_tmpa_int. At the end of the run of digits, check that it ends with a right brace.

```

7130 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
7131 {
7132   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7133   { \l__regex_tmpa_int = \__regex_replacement_g_digits:NN }
7134   { \__regex_replacement_error:NNN g #1 #2 }
7135 }
7136 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
7137 {
7138   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7139   {
7140     \if_int_compare:w \c_one_int < 1#2 \exp_stop_f:
7141     #2
7142     \exp_after:wN \use_i:nnn
7143     \exp_after:wN \__regex_replacement_g_digits:NN
7144   \else:
7145     \exp_stop_f:
7146     \exp_after:wN \__regex_replacement_error:NNN
7147     \exp_after:wN g
7148   \fi:
7149 }
7150 {
7151   \exp_stop_f:
7152   \if_meaning:w \__regex_replacement_rbrace:N #1
7153   \exp_args:No \__regex_replacement_put_submatch:n
7154   { \int_use:N \l__regex_tmpa_int }
7155   \exp_after:wN \use_none:nn
7156   \else:
7157     \exp_after:wN \__regex_replacement_error:NNN
7158     \exp_after:wN g
7159   \fi:
7160 }
7161 #1 #2
7162 }

```

(End of definition for __regex_replacement_g:w and __regex_replacement_g_digits:NN.)

50.6.5 Csnames in replacement

`\__regex_replacement_c:w` `\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```

7163 \cs_new_protected:Npn \__regex_replacement_c:w #1#2
7164 {
7165   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7166   {
7167     \cs_if_exist:cTF { \__regex_replacement_c_#2:w }
7168     { \__regex_replacement_cat:NNN #2 }
7169     { \__regex_replacement_error:NNN c #1#2 }
7170   }
7171   {
7172     \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7173     { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
7174     { \__regex_replacement_error:NNN c #1#2 }
7175   }
7176 }

```

(End of definition for __regex_replacement_c:w.)

`\__regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, protected from expansion by #1 (either `\__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```

7177 \cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
7178 {
7179   \if_case:w \l__regex_replacement_csnames_int
7180   \tl_build_put_right:Nn \l__regex_build_tl
7181   { \exp_not:n { \exp_after:wN #1 \cs:w } }
7182   \else:
7183   \tl_build_put_right:Nn \l__regex_build_tl
7184   { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
7185   \fi:
7186   \int_incr:N \l__regex_replacement_csnames_int
7187 }

```

(End of definition for __regex_replacement_cu_aux:Nw.)

`\__regex_replacement_u:w` Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```

7188 \cs_new_protected:Npn \__regex_replacement_u:w #1#2
7189 {
7190   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7191   { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:V }
7192   { \__regex_replacement_error:NNN u #1#2 }
7193 }

```

(End of definition for __regex_replacement_u:w.)

`\__regex_replacement_rbrace:N` Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

7194 \cs_new_protected:Npn \__regex_replacement_rbrace:N #1
7195 {
7196   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7197     \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
7198     \int_decr:N \l__regex_replacement_csnames_int
7199   \else:
7200     \__regex_replacement_normal:n {#1}
7201   \fi:
7202 }
```

(End of definition for __regex_replacement_rbrace:N.)

`\__regex_replacement_lbrace:N` Within a `\c{...}` or `\u{...}` construction, this is forbidden. Otherwise, this is a raw left brace.

```

7203 \cs_new_protected:Npn \__regex_replacement_lbrace:N #1
7204 {
7205   \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7206     \msg_error:nnn { regex } { cu-lbrace } { u }
7207   \else:
7208     \__regex_replacement_normal:n {#1}
7209   \fi:
7210 }
```

(End of definition for __regex_replacement_lbrace:N.)

50.6.6 Characters in replacement

`\__regex_replacement_cat:NNN` Here, `#1` is a letter among BEMTPUDSLOA and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

7211 \cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
7212 {
7213   \token_if_eq_meaning:NNTF \prg_do_nothing: #3
7214   { \msg_error:nn { regex } { replacement-catcode-end } }
7215   {
7216     \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
7217     {
7218       \msg_error:nnnn
7219       { regex } { replacement-catcode-in-cs } {#1} {#3}
7220       #2 #3
7221     }
7222     {
7223       \__regex_two_if_eq:NNNTF #2 #3 \__regex_replacement_normal:n (
7224       {
7225         \seq_push:NV \l__regex_replacement_category_seq
7226         \l__regex_replacement_category_tl
7227         \tl_set:Nn \l__regex_replacement_category_tl {#1}
7228       }
7229       {
7230         \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
```

```

7231         {
7232             \__regex_char_if_alphanumeric:NTF #3
7233             {
7234                 \msg_error:nnnn
7235                 { regex } { replacement-catcode-escaped }
7236                 {#1} {#3}
7237             }
7238             { }
7239         }
7240         \use:c { __regex_replacement_c_#1:w } #2 #3
7241     }
7242 }
7243 }
7244 }

```

(End of definition for `\__regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```

7245 \group_begin:

```

```

\__regex_replacement_char:nnN

```

The only way to produce an arbitrary character-catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

7246 \cs_new_protected:Npn \__regex_replacement_char:nnN #1#2#3
7247 {
7248     \tex_lccode:D \c_zero_int = '#3 \scan_stop:
7249     \tex_lowercase:D { \__regex_replacement_put:n {#1} }
7250 }

```

(End of definition for `\__regex_replacement_char:nnN`.)

```

\__regex_replacement_c_A:w

```

For an active character, expansion must be avoided, twice because we later do two `e`-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

7251 \char_set_catcode_active:N \^^@
7252 \cs_new_protected:Npn \__regex_replacement_c_A:w
7253 { \__regex_replacement_char:nnN { \exp_not:n { \exp_not:N \^^@ } } }

```

(End of definition for `\__regex_replacement_c_A:w`.)

```

\__regex_replacement_c_B:w

```

An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually `e`-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with `l3tl-analysis`.

```

7254 \char_set_catcode_group_begin:N \^^@
7255 \cs_new_protected:Npn \__regex_replacement_c_B:w
7256 {

```

```

7257     \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7258     \int_incr:N \l__regex_balance_int
7259     \fi:
7260     \__regex_replacement_char:nNN
7261     { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
7262 }

```

(End of definition for __regex_replacement_c_B:w.)

`\__regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two e-expansions.

```

7263     \cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
7264     {
7265         \tl_build_put_right:Nn \l__regex_build_tl
7266         { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
7267     }

```

(End of definition for __regex_replacement_c_C:w.)

`\__regex_replacement_c_D:w` Subscripts fit the mold: `\lowercase` the null byte with the correct category.

```

7268     \char_set_catcode_math_subscript:N ^^@
7269     \cs_new_protected:Npn \__regex_replacement_c_D:w
7270     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_D:w.)

`\__regex_replacement_c_E:w` Similar to the begin-group case, the second e-expansion produces the bare end-group token.

```

7271     \char_set_catcode_group_end:N ^^@
7272     \cs_new_protected:Npn \__regex_replacement_c_E:w
7273     {
7274         \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7275         \int_decr:N \l__regex_balance_int
7276         \fi:
7277         \__regex_replacement_char:nNN
7278         { \exp_not:n { \if_false: { \fi: ^^@ } } }
7279     }

```

(End of definition for __regex_replacement_c_E:w.)

`\__regex_replacement_c_L:w` Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```

7280     \char_set_catcode_letter:N ^^@
7281     \cs_new_protected:Npn \__regex_replacement_c_L:w
7282     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_L:w.)

`\__regex_replacement_c_M:w` No surprise here, we lowercase the null math toggle.

```

7283     \char_set_catcode_math_toggle:N ^^@
7284     \cs_new_protected:Npn \__regex_replacement_c_M:w
7285     { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_M:w.)

`\__regex_replacement_c_0:w` Lowercase an other null byte.

```

7286 \char_set_catcode_other:N \^^@
7287 \cs_new_protected:Npn \__regex_replacement_c_0:w
7288 { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_0:w.)

`\__regex_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two `e`-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```

7289 \char_set_catcode_parameter:N \^^@
7290 \cs_new_protected:Npn \__regex_replacement_c_P:w
7291 {
7292   \__regex_replacement_char:nNN
7293   { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
7294 }

```

(End of definition for __regex_replacement_c_P:w.)

`\__regex_replacement_c_S:w` Spaces are normalized on input by `TEX` to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

7295 \cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
7296 {
7297   \if_int_compare:w '#2 = \c_zero_int
7298     \msg_error:nn { regex } { replacement-null-space }
7299   \fi:
7300   \tex_lccode:D '\ = '#2 \scan_stop:
7301   \tex_lowercase:D { \__regex_replacement_put:n {~} }
7302 }

```

(End of definition for __regex_replacement_c_S:w.)

`\__regex_replacement_c_T:w` No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

7303 \char_set_catcode_alignment:N \^^@
7304 \cs_new_protected:Npn \__regex_replacement_c_T:w
7305 { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_T:w.)

`\__regex_replacement_c_U:w` Simple call to `\__regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```

7306 \char_set_catcode_math_superscript:N \^^@
7307 \cs_new_protected:Npn \__regex_replacement_c_U:w
7308 { \__regex_replacement_char:nNN { ^^@ } }

```

(End of definition for __regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```

7309 \group_end:

```

50.6.7 An error

`\_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
7310 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
7311 {
7312     \msg_error:nne { regex } { replacement-#1 } {#3}
7313     #2 #3
7314 }
```

(End of definition for _regex_replacement_error:NNN.)

50.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
7315 \cs_new_protected:Npn \regex_new:N #1
7316 { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End of definition for \regex_new:N. This function is documented on page 57.)

`\l_tmpa_regex` The usual scratch space.

```
\l_tmpb_regex 7317 \regex_new:N \l_tmpa_regex
\g_tmpa_regex 7318 \regex_new:N \l_tmpb_regex
\g_tmpb_regex 7319 \regex_new:N \g_tmpa_regex
7320 \regex_new:N \g_tmpb_regex
```

(End of definition for \l_tmpa_regex and others. These variables are documented on page 62.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.
`\regex_gset:Nn`
`\regex_const:Nn`

```
7321 \cs_new_protected:Npn \regex_set:Nn #1#2
7322 {
7323     \__regex_compile:n {#2}
7324     \tl_set_eq:NN #1 \l__regex_tmp_regex
7325 }
7326 \cs_new_protected:Npn \regex_gset:Nn #1#2
7327 {
7328     \__regex_compile:n {#2}
7329     \tl_gset_eq:NN #1 \l__regex_tmp_regex
7330 }
7331 \cs_new_protected:Npn \regex_const:Nn #1#2
7332 {
7333     \__regex_compile:n {#2}
7334     \tl_const:Ne #1 { \exp_not:o \l__regex_tmp_regex }
7335 }
```

(End of definition for \regex_set:Nn, \regex_gset:Nn, and \regex_const:Nn. These functions are documented on page 57.)

`\regex_show:n` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `\__regex_show:N` is defined in a different section.
`\regex_log:n`

```
\__regex_show:Nn 7336 \cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\regex_show:N 7337 \cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\regex_log:N 7338 \cs_new_protected:Npn \__regex_show:Nn #1#2
\__regex_show:NN
```



```

7339 {
7340   \__regex_compile:n {#2}
7341   \__regex_show:N \l__regex_tmp_regex
7342   #1 { regex } { show }
7343   { \tl_to_str:n {#2} } { }
7344   { \l__regex_tmpa_tl } { }
7345 }
7346 \cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nneeee }
7347 \cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nneeee }
7348 \cs_new_protected:Npn \__regex_show:NN #1#2
7349 {
7350   \__kernel_chk_tl_type:NnnT #2 { regex }
7351   { \exp_args:No \__regex_clean_regex:n {#2} }
7352   {
7353     \__regex_show:N #2
7354     #1 { regex } { show }
7355     { } { \token_to_str:N #2 }
7356     { \l__regex_tmpa_tl } { }
7357   }
7358 }

```

(End of definition for `\regex_show:n` and others. These functions are documented on page 57.)

`\regex_if_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or false.

```

7359 \prg_new_protected_conditional:Npnn \regex_if_match:nn #1#2 { T , F , TF }
7360 {
7361   \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
7362   \__regex_return:
7363 }
7364 \prg_generate_conditional_variant:Nnn \regex_if_match:nn { nV } { T , F , TF }
7365 \prg_new_protected_conditional:Npnn \regex_if_match:Nn #1#2 { T , F , TF }
7366 {
7367   \__regex_if_match:nn { \__regex_build:N #1 } {#2}
7368   \__regex_return:
7369 }
7370 \prg_generate_conditional_variant:Nnn \regex_if_match:Nn { NV } { T , F , TF }

```

(End of definition for `\regex_if_match:nnTF` and `\regex_if_match:NnTF`. These functions are documented on page 58.)

`\regex_count:nnN` Again, use an auxiliary whose first argument builds the NFA.
`\regex_count:nVN`
`\regex_count:NnN`
`\regex_count:NVN`

```

7371 \cs_new_protected:Npn \regex_count:nnN #1
7372 { \__regex_count:nnN { \__regex_build:n {#1} } }
7373 \cs_new_protected:Npn \regex_count:NnN #1
7374 { \__regex_count:nnN { \__regex_build:N #1 } }
7375 \cs_generate_variant:Nn \regex_count:nnN { nV }
7376 \cs_generate_variant:Nn \regex_count:NnN { NV }

```

(End of definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 58.)

`\regex_match_case:nn`
`\regex_match_case:nnTF`

The auxiliary errors if #1 has an odd number of items, and otherwise it sets `\g__regex_case_int` according to which case was found (zero if not found). The `true` branch leaves the corresponding code in the input stream.

```

7377 \cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
7378 {
7379   \__regex_match_case:nnTF {#1} {#2}
7380   {
7381     \tl_item:nn {#1} { 2 * \g__regex_case_int }
7382     #3
7383   }
7384 }
7385 \cs_new_protected:Npn \regex_match_case:nn #1#2
7386 { \regex_match_case:nnTF {#1} {#2} { } { } }
7387 \cs_new_protected:Npn \regex_match_case:nnT #1#2#3
7388 { \regex_match_case:nnTF {#1} {#2} {#3} { } }
7389 \cs_new_protected:Npn \regex_match_case:nnF #1#2
7390 { \regex_match_case:nnTF {#1} {#2} { } }

```

(End of definition for `\regex_match_case:nnTF`. This function is documented on page 58.)

`\regex_extract_once:nnN`
`\regex_extract_once:nVN`
`\regex_extract_once:nnNTF`
`\regex_extract_once:nVNTF`
`\regex_extract_once:NnN`
`\regex_extract_once:NVN`
`\regex_extract_once:NnNTF`
`\regex_extract_once:NVNTF`
`\regex_extract_all:nnN`
`\regex_extract_all:nVN`
`\regex_extract_all:nnNTF`
`\regex_extract_all:nVNTF`
`\regex_extract_all:NnN`
`\regex_extract_all:NVN`
`\regex_extract_all:NnNTF`
`\regex_extract_all:NVNTF`
`\regex_replace_once:nnN`
`\regex_replace_once:nVN`
`\regex_replace_once:nnNTF`
`\regex_replace_once:nVNTF`
`\regex_replace_once:NnN`
`\regex_replace_once:NVN`
`\regex_replace_once:NnNTF`
`\regex_replace_once:NVNTF`
`\regex_replace_all:nnN`
`\regex_replace_all:nVN`
`\regex_replace_all:nnNTF`
`\regex_replace_all:nVNTF`

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `\__regex_build:n` or `\__regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, etc. The conditionals call `\__regex_return:` to return either true or false once matching has been performed.

```

7391 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
7392 {
7393   \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
7394   \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
7395   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
7396   { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
7397   \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
7398   { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
7399   \cs_generate_variant:Nn #2 { nV }
7400   \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
7401   \cs_generate_variant:Nn #3 { NV }
7402   \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }
7403 }
7404 \__regex_tmp:w \__regex_extract_once:nnN
7405 \regex_extract_once:nnN \regex_extract_once:NnN
7406 \__regex_tmp:w \__regex_extract_all:nnN
7407 \regex_extract_all:nnN \regex_extract_all:NnN
7408 \__regex_tmp:w \__regex_replace_once:nnN
7409 \regex_replace_once:nnN \regex_replace_once:NnN
7410 \__regex_tmp:w \__regex_replace_all:nnN
7411 \regex_replace_all:nnN \regex_replace_all:NnN
7412 \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN

```

(End of definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page 59.)

`\regex_replace_all:NnN`
`\regex_replace_case_once:nnN`
`\regex_replace_all:NVN`
`\regex_replace_case_once:nnNTF`
`\regex_replace_all:NnNTF`
`\regex_replace_all:NVNTF`
`\regex_split:NnN`
`\regex_split:NVN`
`\regex_split:NnNTF`
`\regex_split:NVNTF`
`\regex_split:nnN`
`\regex_split:nVN`
`\regex_split:nnNTF`
`\regex_split:nVNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to

build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

```

7413 \cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
7414 {
7415   \int_if_odd:nTF { \tl_count:n {#1} }
7416   {
7417     \msg_error:nneeee { regex } { case-odd }
7418     { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
7419     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7420     \use_ii:nn
7421   }
7422   {
7423     \__regex_replace_once_aux:nnN
7424     { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7425     { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
7426     #2
7427     \bool_if:NTF \g__regex_success_bool
7428   }
7429 }
7430 \cs_new_protected:Npn \regex_replace_case_once:nN #1#2
7431 { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
7432 \cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
7433 { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
7434 \cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
7435 { \regex_replace_case_once:nNTF {#1} {#2} { } }

```

(End of definition for `\regex_replace_case_once:nNTF`. This function is documented on page 61.)

`\regex_replace_case_all:nN`
`\regex_replace_case_all:nNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_all:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use.

```

7436 \cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
7437 {
7438   \int_if_odd:nTF { \tl_count:n {#1} }
7439   {
7440     \msg_error:nneeee { regex } { case-odd }
7441     { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
7442     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7443     \use_ii:nn
7444   }
7445   {
7446     \__regex_replace_all_aux:nnN
7447     { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7448     { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
7449     #2
7450     \bool_if:NTF \g__regex_success_bool
7451   }
7452 }
7453 \cs_new_protected:Npn \regex_replace_case_all:nN #1#2
7454 { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
7455 \cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
7456 { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
7457 \cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
7458 { \regex_replace_case_all:nNTF {#1} {#2} { } }

```

(End of definition for `\regex_replace_case_all:nNTF`. This function is documented on page 61.)

50.7.1 Variables and helpers for user functions

`\l__regex_match_count_int` The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
7459 \int_new:N \l__regex_match_count_int
```

(End of definition for `\l__regex_match_count_int`.)

`\l__regex_begin_flag` `\l__regex_end_flag` Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```
7460 \flag_new:N \l__regex_begin_flag
```

```
7461 \flag_new:N \l__regex_end_flag
```

(End of definition for `\l__regex_begin_flag` and `\l__regex_end_flag`.)

`\l__regex_min_submatch_int` `\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored in two arrays whose index `<submatch>` ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labeled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
7462 \int_new:N \l__regex_min_submatch_int
```

```
7463 \int_new:N \l__regex_submatch_int
```

```
7464 \int_new:N \l__regex_zeroth_submatch_int
```

(End of definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray` `\g__regex_submatch_begin_intarray` `\g__regex_submatch_end_intarray` `\g__regex_submatch_case_intarray` Hold the place where the match attempt begun, the end-points of each submatch, and which regex case the match corresponds to, respectively.

```
7465 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
```

```
7466 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
```

```
7467 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
```

```
7468 \intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(End of definition for `\g__regex_submatch_prev_intarray` and others.)

`\g__regex_balance_intarray` The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
7469 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(End of definition for `\g__regex_balance_intarray`.)

`\l__regex_added_begin_int` `\l__regex_added_end_int` Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
7470 \int_new:N \l__regex_added_begin_int
```

```
7471 \int_new:N \l__regex_added_end_int
```

(End of definition for `\l__regex_added_begin_int` and `\l__regex_added_end_int`.)

`\__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```

7472 \cs_new_protected:Npn \__regex_return:
7473 {
7474   \if_meaning:w \c_true_bool \g__regex_success_bool
7475   \prg_return_true:
7476   \else:
7477     \prg_return_false:
7478   \fi:
7479 }

```

(End of definition for __regex_return:.)

`\__regex_query_set:n` To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive `\toks` registers. Also store the brace balance (used to check for overall brace balance) in an array.

```

7480 \cs_new_protected:Npn \__regex_query_set:n #1
7481 {
7482   \int_zero:N \l__regex_balance_int
7483   \int_zero:N \l__regex_curr_pos_int
7484   \__regex_query_set_aux:nN { } F
7485   \tl_analysis_map_inline:nn {#1}
7486   { \__regex_query_set_aux:nN {##1} ##3 }
7487   \__regex_query_set_aux:nN { } F
7488   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
7489 }
7490 \cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
7491 {
7492   \int_incr:N \l__regex_curr_pos_int
7493   \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
7494   \__kernel_intarray_gset:Nnn \g__regex_balance_intarray
7495     \l__regex_curr_pos_int \l__regex_balance_int
7496   \if_case:w "#2 \exp_stop_f:
7497   \or: \int_incr:N \l__regex_balance_int
7498   \or: \int_decr:N \l__regex_balance_int
7499   \fi:
7500 }

```

(End of definition for __regex_query_set:n and __regex_query_set_aux:nN.)

50.7.2 Matching

`\__regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```

7501 \cs_new_protected:Npn \__regex_if_match:nn #1#2
7502 {
7503   \group_begin:
7504     \__regex_disable_submatches:
7505     \__regex_single_match:
7506     #1
7507     \__regex_match:n {#2}
7508   \group_end:
7509 }

```

(End of definition for `\__regex_if_match:nn`.)

`\__regex_match_case:nnTF` The code would get badly messed up if the number of items in #1 were not even, so
`\__regex_match_case_aux:nn` we catch this case, then follow the same code as `\regex_if_match:nnTF` but using
`\__regex_case_build:n` and without returning a result.

```

7510 \cs_new_protected:Npn \__regex_match_case:nnTF #1#2
7511   {
7512     \int_if_odd:nTF { \tl_count:n {#1} }
7513     {
7514       \msg_error:nneeee { regex } { case-odd }
7515       { \token_to_str:N \regex_match_case:nn(TF) } { code }
7516       { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7517       \use_i:nn
7518     }
7519     {
7520       \__regex_if_match:nn
7521       { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
7522       {#2}
7523       \bool_if:NTF \g__regex_success_bool
7524     }
7525   }
7526 \cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }

```

(End of definition for `\__regex_match_case:nnTF` and `\__regex_match_case_aux:nn`.)

`\__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first “longest match” is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

7527 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
7528   {
7529     \group_begin:
7530     \__regex_disable_submatches:
7531     \int_zero:N \l__regex_match_count_int
7532     \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
7533     #1
7534     \__regex_match:n {#2}
7535     \exp_args:NNNo
7536     \group_end:
7537     \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
7538   }

```

(End of definition for `\__regex_count:nnN`.)

50.7.3 Extracting submatches

`\__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the
`\__regex_extract_all:nnN` submatches using `\__regex_extract:.`. At the end, store the sequence containing all the
submatches into the user variable #3 after closing the group.

```

7539 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
7540   {
7541     \group_begin:
7542     \__regex_single_match:

```

```

7543     #1
7544     \__regex_match:n {#2}
7545     \__regex_extract:
7546     \__regex_query_set:n {#2}
7547     \__regex_group_end_extract_seq:N #3
7548   }
7549 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
7550 {
7551   \group_begin:
7552     \__regex_multi_match:n { \__regex_extract: }
7553     #1
7554     \__regex_match:n {#2}
7555     \__regex_query_set:n {#2}
7556     \__regex_group_end_extract_seq:N #3
7557   }

```

(End of definition for __regex_extract_once:nnN and __regex_extract_all:nnN.)

__regex_split:nnN Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement \l__regex_submatch_int, which controls which matches will be used.

```

7558 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
7559 {
7560   \group_begin:
7561     \__regex_multi_match:n
7562     {
7563       \if_int_compare:w
7564         \l__regex_start_pos_int < \l__regex_success_pos_int
7565         \__regex_extract:
7566         \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7567         \l__regex_zeroth_submatch_int \c_zero_int
7568         \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7569         \l__regex_zeroth_submatch_int
7570         {
7571           \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
7572           \l__regex_zeroth_submatch_int
7573         }
7574         \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7575         \l__regex_zeroth_submatch_int
7576         \l__regex_start_pos_int
7577       \fi:
7578     }
7579     #1
7580     \__regex_match:n {#2}
7581     \__regex_query_set:n {#2}
7582     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7583     \l__regex_submatch_int \c_zero_int
7584     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7585     \l__regex_submatch_int
7586     \l__regex_max_pos_int

```

```

7587 \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7588 \l__regex_submatch_int
7589 \l__regex_start_pos_int
7590 \int_incr:N \l__regex_submatch_int
7591 \if_meaning:w \c_true_bool \l__regex_empty_success_bool
7592 \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
7593 \int_decr:N \l__regex_submatch_int
7594 \fi:
7595 \fi:
7596 \__regex_group_end_extract_seq:N #3
7597 }

```

(End of definition for __regex_split:nnN.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \g__regex_extract_seq:N \g__regex_tmp_tl, separated by __regex_tmp:w {}. We keep track in the two flags __regex_extract_seq:NNn __regex_begin and __regex_end of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, {} is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by __regex_extract_check:w, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```

7598 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
7599 {
7600   \flag_clear:N \l__regex_begin_flag
7601   \flag_clear:N \l__regex_end_flag
7602   \cs_set_eq:NN \__regex_tmp:w \scan_stop:
7603   \__kernel_tl_gset:Nx \g__regex_tmp_tl
7604   {
7605     \int_step_function:nnN \l__regex_min_submatch_int
7606     { \l__regex_submatch_int - \c_one_int } \__regex_extract_seq_aux:n
7607     \__regex_tmp:w
7608   }
7609   \int_set:Nn \l__regex_added_begin_int
7610   { \flag_height:N \l__regex_begin_flag }
7611   \int_set:Nn \l__regex_added_end_int
7612   { \flag_height:N \l__regex_end_flag }
7613   \tex_afterassignment:D \__regex_extract_check:w
7614   \__kernel_tl_gset:Nx \g__regex_tmp_tl
7615   { \g__regex_tmp_tl \if_false: { \fi: } }
7616   \int_compare:nNnT
7617   { \l__regex_added_begin_int + \l__regex_added_end_int } > \c_zero_int
7618   {
7619     \msg_error:nneee { regex } { result-unbalanced }
7620     { splitting-or-extracting-submatches }
7621     { \int_use:N \l__regex_added_begin_int }
7622     { \int_use:N \l__regex_added_end_int }
7623   }
7624   \group_end:
7625   \__regex_extract_seq:N #1
7626 }
7627 \cs_gset_protected:Npn \__regex_extract_seq:N #1
7628 {

```



```

7629     \seq_clear:N #1
7630     \cs_set_eq:NN \__regex_tmp:w \__regex_extract_seq_loop:Nw
7631     \exp_after:wN \__regex_extract_seq:NNn
7632     \exp_after:wN #1
7633     \g__regex_tmp_tl \use_none:nnn
7634   }
7635   \cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
7636   { #3 #2 #1 \prg_do_nothing: }
7637   \cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
7638   {
7639     \seq_put_right:No #1 {#2}
7640     #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
7641   }

```

(End of definition for __regex_group_end_extract_seq:N and others.)

__regex_extract_seq_aux:n The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

7642   \cs_new:Npn \__regex_extract_seq_aux:n #1
7643   {
7644     \__regex_tmp:w { }
7645     \exp_after:wN \__regex_extract_seq_aux:ww
7646     \int_value:w \__regex_submatch_balance:n {#1} \__regex_sep: #1 \__regex_sep:
7647   }
7648   \cs_new:Npn \__regex_extract_seq_aux:ww #1 \__regex_sep: #2 \__regex_sep:
7649   {
7650     \if_int_compare:w #1 < \c_zero_int
7651     \prg_replicate:nn {-#1}
7652     {
7653       \flag_raise:N \l__regex_begin_flag
7654       \exp_not:n { { \if_false: } \fi: }
7655     }
7656     \fi:
7657     \__regex_query_submatch:n {#2}
7658     \if_int_compare:w #1 > \c_zero_int
7659     \prg_replicate:nn {#1}
7660     {
7661       \flag_raise:N \l__regex_end_flag
7662       \exp_not:n { \if_false: { \fi: } }
7663     }
7664     \fi:
7665   }

```

(End of definition for __regex_extract_seq_aux:n and __regex_extract_seq_aux:ww.)

__regex_extract_check:w In __regex_group_end_extract_seq:N we had to expand \g__regex_tmp_tl to turn \if_false: constructions into actual begin-group and end-group tokens. This is done with a __kernel_tl_gset:Nx assignment, and __regex_extract_check:w is run immediately after this assignment ends, thanks to the \afterassignment primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so }{ is not) then __regex_extract_check:w is called just before the closing brace of the __kernel_tl_gset:Nx (thanks to our sneaky \if_false: { \fi: } construction),

and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `\__kernel_tl_gset:Nx` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```

7666 \cs_new_protected:Npn \__regex_extract_check:w
7667 {
7668   \exp_after:wN \__regex_extract_check:n
7669   \exp_after:wN { \if_false: } \fi:
7670 }
7671 \cs_new_protected:Npn \__regex_extract_check:n #1
7672 {
7673   \tl_if_empty:nF {#1}
7674   {
7675     \int_incr:N \l__regex_added_begin_int
7676     \int_incr:N \l__regex_added_end_int
7677     \tex_afterassignment:D \__regex_extract_check:w
7678     \__kernel_tl_gset:Nx \g__regex_tmp_tl
7679     {
7680       \exp_after:wN \__regex_extract_check_loop:w
7681       \g__regex_tmp_tl
7682       \__regex_tmp:w \__regex_extract_check_end:w
7683       #1
7684     }
7685   }
7686 }
7687 \cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
7688 {
7689   #2
7690   \exp_not:o {#1}
7691   \__regex_tmp:w { }
7692   \__regex_extract_check_loop:w \prg_do_nothing:
7693 }

```

Arguments of `\__regex_extract_check_end:w` are: #1 is the part of the item before the extra end-group token; #2 is junk; #3 is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like #3), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```

7694 \cs_new:Npn \__regex_extract_check_end:w
7695   \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
7696 {
7697   { \exp_not:o {#1} }
7698   #3
7699   \if_false: { \fi: }
7700   \__regex_tmp:w
7701 }

```

(End of definition for `\__regex_extract_check:w` and others.)

`\__regex_extract:` Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. First, we store in `\__regex_extract_aux:w`

\g__regex_submatch_prev_intarray the position at which the match attempt started. We extract the rest from the comma list \l__regex_success_submatches_t1, which starts with entries to be stored in \g__regex_submatch_begin_intarray and continues with entries for \g__regex_submatch_end_intarray.

```

7702 \cs_new_protected:Npn \__regex_extract:
7703 {
7704   \if_meaning:w \c_true_bool \g__regex_success_bool
7705   \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
7706   \prg_replicate:nn \l__regex_capturing_group_int
7707   {
7708     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7709     \l__regex_submatch_int \c_zero_int
7710     \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7711     \l__regex_submatch_int \c_zero_int
7712     \int_incr:N \l__regex_submatch_int
7713   }
7714   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7715   \l__regex_zeroth_submatch_int \l__regex_start_pos_int
7716   \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7717   \l__regex_zeroth_submatch_int \g__regex_case_int
7718   \int_zero:N \l__regex_tmpa_int
7719   \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_t1
7720   \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
7721   \q__regex_recursion_stop
7722   \fi:
7723 }
7724 \cs_new_protected:Npn \__regex_extract_aux:w #1 ,
7725 {
7726   \prg_break: #1 \prg_break_point:
7727   \if_int_compare:w \l__regex_tmpa_int < \l__regex_capturing_group_int
7728     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7729     { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_tmpa_int } {#1}
7730   \else:
7731     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7732     {
7733       \__regex_int_eval:w
7734       \l__regex_zeroth_submatch_int + \l__regex_tmpa_int
7735       - \l__regex_capturing_group_int
7736     }
7737     {#1}
7738   \fi:
7739   \int_incr:N \l__regex_tmpa_int
7740   \__regex_extract_aux:w
7741 }

```

(End of definition for __regex_extract: and __regex_extract_aux:w.)

50.7.4 Replacement

__regex_replace_once:nnN Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the

tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional e-expansion, and checks that braces are balanced in the final result.

```

7742 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2
7743 { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7744 \cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
7745 {
7746   \group_begin:
7747   \__regex_single_match:
7748   #1
7749   \exp_args:No \__regex_match:n {#3}
7750   \bool_if:NTF \g__regex_success_bool
7751   {
7752     \__regex_extract:
7753     \exp_args:No \__regex_query_set:n {#3}
7754     #2
7755     \int_set:Nn \l__regex_balance_int
7756     { \__regex_replacement_balance_one_match:n \l__regex_zeroth_submatch_int }
7757     \__kernel_tl_set:Nx \l__regex_tmpa_tl
7758     {
7759       \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
7760       \__regex_query_range:nn
7761       {
7762         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7763         \l__regex_zeroth_submatch_int
7764       }
7765       \l__regex_max_pos_int
7766     }
7767     \__regex_group_end_replace:N #3
7768   }
7769   { \group_end: }
7770 }

```

(End of definition for `\__regex_replace_once:nnN` and `\__regex_replace_once_aux:nnN`.)

`\__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

7771 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2
7772 { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7773 \cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
7774 {
7775   \group_begin:
7776   \__regex_multi_match:n { \__regex_extract: }
7777   #1
7778   \exp_args:No \__regex_match:n {#3}
7779   \exp_args:No \__regex_query_set:n {#3}

```

```

7780     #2
7781     \int_set:Nn \l__regex_balance_int
7782     {
7783         \c_zero_int
7784         \int_step_function:nnnN
7785         \l__regex_min_submatch_int
7786         \l__regex_capturing_group_int
7787         { \l__regex_submatch_int - \c_one_int }
7788         \__regex_replacement_balance_one_match:n
7789     }
7790     \__kernel_tl_set:Nx \l__regex_tmpa_tl
7791     {
7792         \int_step_function:nnnN
7793         \l__regex_min_submatch_int
7794         \l__regex_capturing_group_int
7795         { \l__regex_submatch_int - \c_one_int }
7796         \__regex_replacement_do_one_match:n
7797         \__regex_query_range:nn
7798         \l__regex_start_pos_int \l__regex_max_pos_int
7799     }
7800     \__regex_group_end_replace:N #3
7801 }

```

(End of definition for __regex_replace_all:nnN.)

__regex_group_end_replace:N At this stage \l__regex_tmpa_tl (e-expands to the desired result). Guess from \l__-
 __regex_group_end_replace_try: regex_balance_int the number of braces to add before or after the result then try
 __regex_group_end_replace_check:w expanding. The simplest case is when \l__regex_tmpa_tl together with the braces we
 __regex_group_end_replace_check:n insert via \prg_replicate:nn give a balanced result, and the assignment ends at the
 \if_false: { \fi: } construction: then __regex_group_end_replace_check:w sees that there is no material left and we successfully found the result. The harder case
 is that expanding \l__regex_tmpa_tl may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has
 not yet been expanded so that __regex_group_end_replace_check:n grabs everything until the last brace in __regex_group_end_replace_try:, letting us try again with an
 extra surrounding pair of braces.

```

7802 \cs_new_protected:Npn \__regex_group_end_replace:N #1
7803 {
7804     \int_set:Nn \l__regex_added_begin_int
7805     { \int_max:nn { - \l__regex_balance_int } \c_zero_int }
7806     \int_set:Nn \l__regex_added_end_int
7807     { \int_max:nn \l__regex_balance_int \c_zero_int }
7808     \__regex_group_end_replace_try:
7809     \int_compare:nNtT { \l__regex_added_begin_int + \l__regex_added_end_int }
7810     > \c_zero_int
7811     {
7812         \msg_error:nneee { regex } { result-unbalanced }
7813         { replacing } { \int_use:N \l__regex_added_begin_int }
7814         { \int_use:N \l__regex_added_end_int }
7815     }
7816     \group_end:
7817     \tl_set_eq:NN #1 \g__regex_tmpa_tl
7818 }

```

```

7819 \cs_new_protected:Npn \__regex_group_end_replace_try:
7820 {
7821   \tex_afterassignment:D \__regex_group_end_replace_check:w
7822   \__kernel_tl_gset:Nx \g__regex_tmp_tl
7823   {
7824     \prg_replicate:nn \l__regex_added_begin_int { { \if_false: } \fi: }
7825     \l__regex_tmpa_tl
7826     \prg_replicate:nn \l__regex_added_end_int { \if_false: { \fi: } }
7827     \if_false: { \fi: }
7828   }
7829 }
7830 \cs_new_protected:Npn \__regex_group_end_replace_check:w
7831 {
7832   \exp_after:wN \__regex_group_end_replace_check:n
7833   \exp_after:wN { \if_false: } \fi:
7834 }
7835 \cs_new_protected:Npn \__regex_group_end_replace_check:n #1
7836 {
7837   \tl_if_empty:nF {#1}
7838   {
7839     \int_incr:N \l__regex_added_begin_int
7840     \int_incr:N \l__regex_added_end_int
7841     \__regex_group_end_replace_try:
7842   }
7843 }

```

(End of definition for __regex_group_end_replace:N and others.)

50.7.5 Peeking ahead

\l__regex_peek_true_tl True/false code arguments of \peek_regex:nTF or similar.

```

7844 \tl_new:N \l__regex_peek_true_tl
7845 \tl_new:N \l__regex_peek_false_tl

```

(End of definition for \l__regex_peek_true_tl and \l__regex_peek_false_tl.)

\l__regex_replacement_tl When peeking in \peek_regex_replace_once:nnTF we need to store the replacement text.

```

7846 \tl_new:N \l__regex_replacement_tl

```

(End of definition for \l__regex_replacement_tl.)

\l__regex_input_tl Stores each token found as __regex_input_item:n {<tokens>}, where the <tokens> o-expand to the token found, as for \tl_analysis_map_inline:nn.

```

7847 \tl_new:N \l__regex_input_tl
7848 \cs_new_eq:NN \__regex_input_item:n ?

```

(End of definition for \l__regex_input_tl and __regex_input_item:n.)

\peek_regex:nTF

\peek_regex:NTF

\peek_regex_remove_once:nTF

\peek_regex_remove_once:NTF

The T and F functions just call the corresponding TF function. The four TF functions differ along two axes: whether to remove the token or not, distinguished by using __regex_peek_end: or __regex_peek_remove_end:n (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an N-type variable, distinguished by calling __regex_build_aux:Nn or __regex_build_aux:NN. The first

argument of these functions is `\c_false_bool` to indicate that there should be no implicit insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```

7849 \cs_new_protected:Npn \peek_regex:nTF #1
7850 {
7851   \__regex_peek:nnTF
7852     { \__regex_build_aux:Nn \c_false_bool {#1} }
7853     { \__regex_peek_end: }
7854 }
7855 \cs_new_protected:Npn \peek_regex:nT #1#2
7856 { \peek_regex:nTF {#1} {#2} { } }
7857 \cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
7858 \cs_new_protected:Npn \peek_regex:NTF #1
7859 {
7860   \__regex_peek:nnTF
7861     { \__regex_build_aux:NN \c_false_bool #1 }
7862     { \__regex_peek_end: }
7863 }
7864 \cs_new_protected:Npn \peek_regex:NT #1#2
7865 { \peek_regex:NTF #1 {#2} { } }
7866 \cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
7867 \cs_new_protected:Npn \peek_regex_remove_once:nTF #1
7868 {
7869   \__regex_peek:nnTF
7870     { \__regex_build_aux:Nn \c_false_bool {#1} }
7871     { \__regex_peek_remove_end:n {##1} }
7872 }
7873 \cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
7874 { \peek_regex_remove_once:nTF {#1} {#2} { } }
7875 \cs_new_protected:Npn \peek_regex_remove_once:nF #1
7876 { \peek_regex_remove_once:nTF {#1} { } }
7877 \cs_new_protected:Npn \peek_regex_remove_once:NTF #1
7878 {
7879   \__regex_peek:nnTF
7880     { \__regex_build_aux:NN \c_false_bool #1 }
7881     { \__regex_peek_remove_end:n {##1} }
7882 }
7883 \cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
7884 { \peek_regex_remove_once:NTF #1 {#2} { } }
7885 \cs_new_protected:Npn \peek_regex_remove_once:NF #1
7886 { \peek_regex_remove_once:NTF #1 { } }

```

(End of definition for `\peek_regex:nTF` and others. These functions are documented on page 215.)

`\__regex_peek:nnTF`
`\__regex_peek_aux:nnTF`

Store the user's true/false codes (plus `\group_end:`) into two token lists. Then build the automaton with `#1`, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_if_match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `\__regex_match_one_token:nnN` calls `\__regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```

7887 \cs_new_protected:Npn \__regex_peek:nnTF #1

```

```

7888 {
7889   \__regex_peek_aux:nnTF
7890   {
7891     \__regex_disable_submatches:
7892     #1
7893   }
7894 }
7895 \cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
7896 {
7897   \group_begin:
7898   \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
7899   \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
7900   \__regex_single_match:
7901   #1
7902   \__regex_match_init:
7903   \tl_build_begin:N \l__regex_input_tl
7904   \__regex_match_once_init:
7905   \peek_analysis_map_inline:n
7906   {
7907     \tl_build_put_right:Nn \l__regex_input_tl
7908     { \__regex_input_item:n {##1} }
7909     \__regex_match_one_token:nnN {##1} {##2} ##3
7910     \use_none:nnn
7911     \prg_break_point:Nn \__regex_maplike_break:
7912     { \peek_analysis_map_break:n {#2} }
7913   }
7914 }

```

(End of definition for __regex_peek:nnTF and __regex_peek_aux:nnTF.)

__regex_peek_end: Once the regex matches (or permanently fails to match) we call __regex_peek_end:, or
 __regex_peek_remove_end:n __regex_peek_remove_end:n with argument the last token seen. For \peek_regex:nTF
 we reinsert tokens seen by calling __regex_peek_reinsert:N regardless of the result
 of the match. For \peek_regex_remove_once:nTF we reinsert the tokens seen only if
 the match failed; otherwise we just reinsert the tokens #1, with one expansion. To be
 more precise, #1 consists of tokens that o-expand and e-expand to the last token seen, for
 example it is \exp_not:n \{ <cs> \} for a non-expandable primitive. Thanks to \exp_
 not:n (rather than \exp_not:N) we don't need to worry about the suppressed further
 expansion of <cs>.

```

7915 \cs_new_protected:Npn \__regex_peek_end:
7916 {
7917   \bool_if:NTF \g__regex_success_bool
7918   { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
7919   { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7920 }
7921 \cs_new_protected:Npn \__regex_peek_remove_end:n #1
7922 {
7923   \bool_if:NTF \g__regex_success_bool
7924   { \exp_after:wN \l__regex_peek_true_tl #1 }
7925   { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7926 }

```

(End of definition for __regex_peek_end: and __regex_peek_remove_end:n.)

`\__regex_peek_reinsert:N` Insert the true/false code #1, followed by the tokens found, which were stored in `\l__regex_input_tl`. For this, loop through that token list using `\__regex_reinsert_item:n`, which expands #1 once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:`. We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```

7927 \cs_new_protected:Npn \__regex_peek_reinsert:N #1
7928 {
7929     \tl_build_end:N \l__regex_input_tl
7930     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7931     \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
7932 }
7933 \cs_new_protected:Npn \__regex_reinsert_item:n #1
7934 {
7935     \exp_after:wN \exp_after:wN
7936     \exp_after:wN \exp_end:
7937     \exp_after:wN \exp_after:wN
7938     #1
7939     \exp:w
7940 }

```

(End of definition for `\__regex_peek_reinsert:N` and `\__regex_reinsert_item:n`.)

`\peek_regex_replace_once:nn` Similar to `\peek_regex:nTF` above.

`\peek_regex_replace_once:nnTF`

`\peek_regex_replace_once:Nn`

`\peek_regex_replace_once:NnTF`

```

7941 \cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
7942 { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
7943 \cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
7944 { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
7945 \cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
7946 { \peek_regex_replace_once:nnTF {#1} {#2} { } }
7947 \cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
7948 { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
7949 \cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
7950 { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
7951 \cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
7952 { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
7953 \cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
7954 { \peek_regex_replace_once:NnTF #1 {#2} { } }
7955 \cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
7956 { \peek_regex_replace_once:NnTF #1 {#2} { } { } }

```

(End of definition for `\peek_regex_replace_once:nnTF` and `\peek_regex_replace_once:NnTF`. These functions are documented on page 216.)

`\__regex_peek_replace:nnTF` Same as `\__regex_peek:nnTF` (used for `\peek_regex:nTF` above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```

7957 \cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
7958 {
7959     \tl_set:Nn \l__regex_replacement_tl {#2}
7960     \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
7961 }

```

(End of definition for `\__regex_peek_replace:nnTF`.)

`\__regex_peek_replace_end:` If the match failed `\__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `\__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behavior as explained below. Analyze the replacement text with `\__regex_replacement:n`, which as usual defines `\__regex_replacement_do_one_match:n` to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:e` expands for instance the trailing `\__regex_query_range:nn` down to a sequence of `\__regex_reinsert_item:n {⟨tokens⟩}` where `⟨tokens⟩` o-expand to a single token that we want to insert. After e-expansion, `\use:e` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:.` This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```

7962 \cs_new_protected:Npn \__regex_peek_replace_end:
7963 {
7964   \bool_if:NTF \g__regex_success_bool
7965   {
7966     \__regex_extract:
7967     \__regex_query_set_from_input_tl:
7968     \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
7969     \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
7970     \__regex_peek_replacement_put_submatch_aux:n
7971     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
7972     \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
7973     \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
7974     \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
7975     \use:e
7976     {
7977       \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
7978       \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
7979       \__regex_query_range:nn
7980       {
7981         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7982         \l__regex_zeroth_submatch_int
7983       }
7984       \l__regex_max_pos_int
7985       \exp_end:
7986     }
7987   }
7988   { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
7989 }

```

(End of definition for `\__regex_peek_replace_end:.`)

`\__regex_query_set_from_input_tl:` The input was stored into `\l__regex_input_tl` as successive items `\__regex_input_item:n {⟨tokens⟩}`. Store that in successive `\toks`. It's not clear whether the empty entries before and after are both useful.

```

7990 \cs_new_protected:Npn \__regex_query_set_from_input_tl:
7991 {
7992   \tl_build_end:N \l__regex_input_tl
7993   \int_zero:N \l__regex_curr_pos_int
7994   \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
7995   \__regex_query_set_item:n { }
7996   \l__regex_input_tl

```

```

7997     \__regex_query_set_item:n { }
7998     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
7999   }
8000 \cs_new_protected:Npn \__regex_query_set_item:n #1
8001 {
8002     \int_incr:N \l__regex_curr_pos_int
8003     \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
8004 }

```

(End of definition for __regex_query_set_from_input_tl: and __regex_query_set_item:n.)

__regex_peek_replacement_put:n

While building the replacement function __regex_replacement_do_one_match:n, we often want to put simple material, given as #1, whose e-expansion o-expands to a single token. Normally we can just add the token to \l__regex_build_tl, but for \peek_regex_replace_once:nnTF we eventually want to do some strange expansion that is basically using \exp_after:wN to jump through numerous tokens (we cannot use e-expansion like for \regex_replace_once:nnTF because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because \cs:w ... \cs_end: does all the expansion we need.

```

8005 \cs_new_protected:Npn \__regex_peek_replacement_put:n #1
8006 {
8007     \if_case:w \l__regex_replacement_csnames_int
8008     \tl_build_put_right:Nn \l__regex_build_tl
8009     { \exp_not:N \__regex_reinsert_item:n {#1} }
8010     \else:
8011     \tl_build_put_right:Nn \l__regex_build_tl {#1}
8012     \fi:
8013 }

```

(End of definition for __regex_peek_replacement_put:n.)

__regex_peek_replacement_token:n

When hit with \exp:w, __regex_peek_replacement_token:n {<token>} stops \exp_end: and does \exp_after:wN <token> \exp:w to continue expansion after it.

```

8014 \cs_new_protected:Npn \__regex_peek_replacement_token:n #1
8015 { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }

```

(End of definition for __regex_peek_replacement_token:n.)

__regex_peek_replacement_put_submatch_aux:n

While analyzing the replacement we also have to insert submatches found in the query. Since query items __regex_input_item:n {<tokens>} expand correctly only when surrounded by \exp:w ... \exp_end:, and since these expansion controls are not there within csnames (because \cs:w ... \cs_end: make them unnecessary in most cases), we have to put \exp:w and \exp_end: by hand here.

```

8016 \cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
8017 {
8018     \if_case:w \l__regex_replacement_csnames_int
8019     \tl_build_put_right:Nn \l__regex_build_tl
8020     { \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
8021     \else:
8022     \tl_build_put_right:Nn \l__regex_build_tl
8023     {
8024         \exp:w
8025         \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: }

```

```

8026         \exp_end:
8027     }
8028     \fi:
8029 }

```

(End of definition for `\_regex_peek_replacement_put_submatch_aux:n`.)

`\_regex_peek_replacement_var:N` This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable #1 and stopping the `\exp:w` that precedes.

```

8030 \cs_new_protected:Npn \_regex_peek_replacement_var:N #1
8031 {
8032     \exp_after:wN \exp_last_unbraced:NV
8033     \exp_after:wN \exp_end:
8034     \exp_after:wN #1
8035     \exp:w
8036 }

```

(End of definition for `\_regex_peek_replacement_var:N`.)

50.8 Messages

Messages for the preparsing phase.

```

8037 \use:e
8038 {
8039     \msg_new:nnn { regex } { trailing-backslash }
8040     { Trailing~'\iow_char:N\'\~in-regex-or-replacement. }
8041     \msg_new:nnn { regex } { x-missing-rbrace }
8042     {
8043         Missing-brace~'\iow_char:N\}\~in-regex~
8044         '...\iow_char:N\x\iow_char:N\{...#1'.
8045     }
8046     \msg_new:nnn { regex } { x-overflow }
8047     {
8048         Character-code~##1~too-large-in~
8049         \iow_char:N\x\iow_char:N\{##2\iow_char:N\}-regex.
8050     }
8051 }

```

Invalid quantifier.

```

8052 \msg_new:nnnn { regex } { invalid-quantifier }
8053 { Braced-quantifier~'#1'~may-not-be-followed-by~'#2'. }
8054 {
8055     The-character~'#2'~is~invalid~in~the-braced-quantifier~'#1'.~
8056     The-only-valid-quantifiers-are~'*',~'?',~'+',~'{<int>}',~
8057     '{<min>}',~and~'{<min>,<max>}',~optionally~followed-by~'?''.
8058 }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

8059 \msg_new:nnnn { regex } { missing-rbrack }
8060 { Missing-right-bracket-inserted-in-regular-expression. }
8061 {
8062     LaTeX-was-given-a-regular-expression-where-a-character-class~
8063     was-started-with~'[',~but~the~matching~']'~is-missing.

```

```

8064 }
8065 \msg_new:nnnn { regex } { missing-rparen }
8066 {
8067   Missing-right~
8068   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
8069   inserted-in-regular-expression.
8070 }
8071 {
8072   LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
8073   more-left-parentheses-than-right-parentheses.
8074 }
8075 \msg_new:nnnn { regex } { extra-rparen }
8076 { Extra-right-parenthesis-ignored-in-regular-expression. }
8077 {
8078   LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
8079   was-open.~The-parenthesis-will-be-ignored.
8080 }

```

Some escaped alphanumerics are not allowed everywhere.

```

8081 \msg_new:nnnn { regex } { bad-escape }
8082 {
8083   Invalid-escape~'\iow_char:N\\#1'~
8084   \__regex_if_in_cs:TF { within-a-control-sequence. }
8085   {
8086     \__regex_if_in_class:TF
8087     { in-a-character-class. }
8088     { following-a-category-test. }
8089   }
8090 }
8091 {
8092   The-escape-sequence~'\iow_char:N\\#1'~may-not-appear~
8093   \__regex_if_in_cs:TF
8094   {
8095     within-a-control-sequence-test-introduced-by~
8096     '\iow_char:N\\c\iow_char:N\{' .
8097   }
8098   {
8099     \__regex_if_in_class:TF
8100     { within-a-character-class~ }
8101     { following-a-category-test-such-as~'\iow_char:N\\cL'~ }
8102     because-it-does-not-match-exactly-one-character.
8103   }
8104 }

```

Range errors.

```

8105 \msg_new:nnnn { regex } { range-missing-end }
8106 { Invalid-end-point-for-range~'#1-#2'~in-character-class. }
8107 {
8108   The-end-point~'#2'~of-the-range~'#1-#2'~may-not-serve-as-an~
8109   end-point-for-a-range:~alphanumeric-characters-should-not-be~
8110   escaped,~and-non-alphanumeric-characters-should-be-escaped.
8111 }
8112 \msg_new:nnnn { regex } { range-backwards }
8113 { Range~'#1-#2'~out-of-order-in-character-class. }
8114 {

```

```

8115     In~ranges~of~characters~'[x-y]~'~appearing~in~character~classes,~
8116     the~first~character~code~must~not~be~larger~than~the~second.~
8117     Here,~'#1'~has~character~code~\int_eval:n~{'#1},~while~
8118     '#2'~has~character~code~\int_eval:n~{'#2}.
8119 }
      Errors related to \c and \u.
8120 \msg_new:nnnn { regex } { c-bad-mode }
8121 { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
8122 {
8123     The~'\iow_char:N\\c'~escape~cannot~be~used~within~
8124     a~control~sequence~test~'\iow_char:N\\c{...}'~
8125     nor~another~category~test.~
8126     To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
8127 }
8128 \msg_new:nnnn { regex } { c-C-invalid }
8129 { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'('~,~not~'#1'. }
8130 {
8131     The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
8132     control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
8133     It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
8134 }
8135 \msg_new:nnnn { regex } { cu-lbrace }
8136 { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}'. }
8137 {
8138     Constructions~such~as~'\iow_char:N\\#1{...\\iow_char:N\\{...}'~are~
8139     not~allowed~and~should~be~replaced~by~
8140     '\iow_char:N\\#1{...\\token_to_str:N\\{...}'.
8141 }
8142 \msg_new:nnnn { regex } { c-lparen-in-class }
8143 { Catcode~test~cannot~apply~to~group~in~character~class }
8144 {
8145     Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
8146     class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
8147 }
8148 \msg_new:nnnn { regex } { c-missing-rbrace }
8149 { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
8150 {
8151     LaTeX~was~given~a~regular~expression~where~a~
8152     '\iow_char:N\\c\\iow_char:N\\{...}'~construction~was~not~ended~
8153     with~a~closing~brace~'\iow_char:N\\}'.
8154 }
8155 \msg_new:nnnn { regex } { c-missing-rbrack }
8156 { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
8157 {
8158     A~construction~'\iow_char:N\\c[...]'~appears~in~a~
8159     regular~expression,~but~the~closing~'~'~is~not~present.
8160 }
8161 \msg_new:nnnn { regex } { c-missing-category }
8162 { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
8163 {
8164     In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
8165     may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
8166     capital~letter~representing~a~character~category,~namely~
8167     one~of~'ABCDELMOPTU'.

```

```

8168 }
8169 \msg_new:nnnn { regex } { c-trailing }
8170 { Trailing-category-code-escape~'\iow_char:N\c'... }
8171 {
8172   A~regular~expression~ends~with~'\iow_char:N\c'~followed~
8173   by~a~letter.~It~will~be~ignored.
8174 }
8175 \msg_new:nnnn { regex } { u-missing-lbrace }
8176 { Missing~left~brace~following~'\iow_char:N\{~escape. }
8177 {
8178   The~'\iow_char:N\{~escape~sequence~must~be~followed~by~
8179   a~brace~group~with~the~name~of~the~variable~to~use.
8180 }
8181 \msg_new:nnnn { regex } { u-missing-rbrace }
8182 { Missing~right~brace~inserted~for~'\iow_char:N\}~escape. }
8183 {
8184   LaTeX~
8185   \str_if_eq:eeTF { } {#2}
8186   { reached~the~end~of~the~string~ }
8187   { encountered~an~escaped~alphanumeric~character '\iow_char:N\#2'~ }
8188   when~parsing~the~argument~of~an~
8189   '\iow_char:N\{~\iow_char:N\{...}\}'~escape.
8190 }

```

Errors when encountering the POSIX syntax [:...:].

```

8191 \msg_new:nnnn { regex } { posix-unsupported }
8192 { POSIX~collating~element~'[#1 ~ #1]~not~supported. }
8193 {
8194   The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
8195   in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
8196   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
8197 }
8198 \msg_new:nnnn { regex } { posix-unknown }
8199 { POSIX~class~'[:#1:]'~unknown. }
8200 {
8201   '[:#1:]'~is~not~among~the~known~POSIX~classes~
8202   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
8203   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
8204   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
8205   '[:word:]',~and~'[:xdigit:]'.
8206 }
8207 \msg_new:nnnn { regex } { posix-missing-close }
8208 { Missing~closing~':'~for~POSIX~class. }
8209 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

8210 \msg_new:nnnn { regex } { result-unbalanced }
8211 { Missing~brace~inserted~when~#1. }
8212 {
8213   LaTeX~was~asked~to~do~some~regular~expression~operation,~
8214   and~the~resulting~token~list~would~not~have~the~same~number~
8215   of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
8216   #2~left,~#3~right.

```

```

8217 }
      Error message for unknown options.
8218 \msg_new:nnnn { regex } { unknown-option }
8219 { Unknown~option~'#1'~for~regular~expressions. }
8220 {
8221   The~only~available~option~is~'case-insensitive',~toggled~by~
8222   '(?i)'~and~'(?-i)'.
8223 }
8224 \msg_new:nnnn { regex } { special-group-unknown }
8225 { Unknown~special~group~'#1~...'~in~a~regular~expression. }
8226 {
8227   The~only~valid~constructions~starting~with~'(?~are~
8228   '(:~...'~)',~'(?|~...'~)',~'(?i)',~and~'(?-i)'.
8229 }
      Errors in the replacement text.
8230 \msg_new:nnnn { regex } { replacement-c }
8231 { Misused~'\iow_char:N\c'~command~in~a~replacement~text. }
8232 {
8233   In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8234   can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
8235   or~a~brace~group,~not~by~'#1'.
8236 }
8237 \msg_new:nnnn { regex } { replacement-u }
8238 { Misused~'\iow_char:N\u'~command~in~a~replacement~text. }
8239 {
8240   In~a~replacement~text,~the~'\iow_char:N\u'~escape~sequence~
8241   must~be~followed~by~a~brace~group~holding~the~name~of~the~
8242   variable~to~use.
8243 }
8244 \msg_new:nnnn { regex } { replacement-g }
8245 {
8246   Missing~brace~for~the~'\iow_char:N\g'~construction~
8247   in~a~replacement~text.
8248 }
8249 {
8250   In~the~replacement~text~for~a~regular~expression~search,~
8251   submatches~are~represented~either~as~'\iow_char:N \g{dd..d}',~
8252   or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
8253 }
8254 \msg_new:nnnn { regex } { replacement-catcode-end }
8255 {
8256   Missing~character~for~the~'\iow_char:N\c<category><character>'~
8257   construction~in~a~replacement~text.
8258 }
8259 {
8260   In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8261   can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
8262   the~character~category.~Then,~a~character~must~follow.~LaTeX~
8263   reached~the~end~of~the~replacement~when~looking~for~that.
8264 }
8265 \msg_new:nnnn { regex } { replacement-catcode-escaped }
8266 {
8267   Escaped~letter~or~digit~after~category~code~in~replacement~text.

```



```

8268 }
8269 {
8270   In~a~replacement~text,~the~'\iow_char:N\c'~escape~sequence~
8271   can~be~followed~by~one~of~the~letters~'ABCDELMOPTU'~representing~
8272   the~character~category.~Then,~a~character~must~follow,~not~
8273   '\iow_char:N\#2'.
8274 }
8275 \msg_new:nnnn { regex } { replacement-catcode-in-cs }
8276 {
8277   Category~code~'\iow_char:N\c#1#3'~ignored~inside~
8278   '\iow_char:N\c\{...\}'~in~a~replacement~text.
8279 }
8280 {
8281   In~a~replacement~text,~the~category~codes~of~the~argument~of~
8282   '\iow_char:N\c\{...\}'~are~ignored~when~building~the~control~
8283   sequence~name.
8284 }
8285 \msg_new:nnnn { regex } { replacement-null-space }
8286 { TeX~cannot~build~a~space~token~with~character~code~0. }
8287 {
8288   You~asked~for~a~character~token~with~category~space,~
8289   and~character~code~0,~for~instance~through~
8290   '\iow_char:N\cS\iow_char:N\#x00'.~
8291   This~specific~case~is~impossible~and~will~be~replaced~
8292   by~a~normal~space.
8293 }
8294 \msg_new:nnnn { regex } { replacement-missing-rbrace }
8295 { Missing~right~brace~inserted~in~replacement~text. }
8296 {
8297   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8298   missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
8299 }
8300 \msg_new:nnnn { regex } { replacement-missing-rparen }
8301 { Missing~right~parenthesis~inserted~in~replacement~text. }
8302 {
8303   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8304   missing~right~
8305   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
8306 }
8307 \msg_new:nnn { regex } { submatch-too-big }
8308 { Submatch~#1~used~but~regex~only~has~#2~group(s) }
      Some escaped alphanumerics are not allowed everywhere.
8309 \msg_new:nnnn { regex } { backwards-quantifier }
8310 { Quantifier~"{#1,#2}"~is~backwards. }
8311 { The~values~given~in~a~quantifier~must~be~in~order. }
      Used in user commands, and when showing a regex.
8312 \msg_new:nnnn { regex } { case-odd }
8313 { #1~with~odd~number~of~items }
8314 {
8315   There~must~be~a~#2~part~for~each~regex:~
8316   found~odd~number~of~items~(#3)~in~\
8317   \iow_indent:n {#4}
8318 }

```

```

8319 \msg_new:nnn { regex } { show }
8320 {
8321   >~Compiled~regex~
8322   \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
8323   #3
8324 }
8325 \prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
8326 \prop_gput:Nnn \g_msg_module_type_prop { regex } { }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: **#1** is the minimum number of repetitions; **#2** is the number of allowed extra repetitions (-1 for infinite number), and **#3** tells us about laziness.

```

8327 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
8328 {
8329   \str_if_eq:eeF { #1 #2 } { 1 0 }
8330   {
8331     , ~ repeated ~
8332     \int_case:nnF {#2}
8333     {
8334       { -1 } { #1~or-more-times,~\bool_if:NTF #3 { lazy } { greedy } }
8335       { 0 } { #1~times }
8336     }
8337     {
8338       between~#1~and~\int_eval:n {#1+#2}~times,~
8339       \bool_if:NTF #3 { lazy } { greedy }
8340     }
8341   }
8342 }

```

(End of definition for `\__regex_msg_repeated:nnN`.)

50.9 Code for tracing

There is a more extensive implementation of tracing in the `l3trial` package `l3trace`. Function names are a bit different but could be merged.

`__regex_trace_push:nnN` Here **#1** is the module name (`regex`) and **#2** is typically 1. If the module's current tracing level is less than **#2** show nothing, otherwise write **#3** to the terminal.

`__regex_trace_pop:nnN`

`__regex_trace:nne`

```

8343 \cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
8344 { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
8345 \cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
8346 { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
8347 \cs_new_protected:Npn \__regex_trace:nne #1#2#3
8348 {
8349   \int_compare:nNnF
8350   { \int_use:c { g__regex_trace_#1_int } } < {#2}
8351   { \iow_term:e { Trace:~#3 } }
8352 }

```

(End of definition for `\__regex_trace_push:nnN`, `\__regex_trace_pop:nnN`, and `\__regex_trace:nne`.)

`g__regex_trace_regex_int` No tracing when that is zero.

```

8353 \int_new:N \g__regex_trace_regex_int

```

(End of definition for \g__regex_trace_regex_int.)

__regex_trace_states:n This function lists the contents of all states of the NFA, stored in \toks from 0 to \l__-
regex_max_state_int (excluded).

```
8354 \cs_new_protected:Npn \__regex_trace_states:n #1
8355 {
8356   \int_step_inline:nnn
8357     \l__regex_min_state_int
8358     { \l__regex_max_state_int - \c_one_int }
8359     {
8360       \__regex_trace:nne { regex } {#1}
8361       { \iow_char:N \toks ##1 = { \__regex_toks_use:w ##1 } }
8362     }
8363 }
```

(End of definition for __regex_trace_states:n.)

8364 `</code>`

Chapter 51

l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

```
8365 \*code>
```

51.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in `l3basics`, as it's needed earlier to define quark test functions.

```
8366 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(End of definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 74.)

51.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

(End of definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 66.)

51.3 The boolean data type

```
8367 <@@=bool>
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
8368 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
8369 \cs_generate_variant:Nn \bool_new:N { c }
```

(End of definition for `\bool_new:N`. This function is documented on page 68.)

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
\bool_const:cn
8370 \cs_new_protected:Npn \bool_const:Nn #1#2
8371 {
8372   \__kernel_chk_if_free_cs:N #1
8373   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
8374 }
8375 \cs_generate_variant:Nn \bool_const:Nn { c }
```

(End of definition for `\bool_const:Nn`. This function is documented on page 68.)

```

\bool_set_true:N Setting is already pretty easy. When check-declarations is active, the definitions are
\bool_set_true:c patched to make sure the boolean exists. This is needed because booleans are not based
\bool_gset_true:N on token lists nor on TEX registers.
\bool_gset_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c
8376 \cs_new_protected:Npn \bool_set_true:N #1
8377 { \cs_set_eq:NN #1 \c_true_bool }
8378 \cs_new_protected:Npn \bool_set_false:N #1
8379 { \cs_set_eq:NN #1 \c_false_bool }
8380 \cs_new_protected:Npn \bool_gset_true:N #1
8381 { \cs_gset_eq:NN #1 \c_true_bool }
8382 \cs_new_protected:Npn \bool_gset_false:N #1
8383 { \cs_gset_eq:NN #1 \c_false_bool }
8384 \cs_generate_variant:Nn \bool_set_true:N { c }
8385 \cs_generate_variant:Nn \bool_set_false:N { c }
8386 \cs_generate_variant:Nn \bool_gset_true:N { c }
8387 \cs_generate_variant:Nn \bool_gset_false:N { c }

```

(End of definition for `\bool_set_true:N` and others. These functions are documented on page 68.)

```

\bool_set_eq:NN The usual copy code. While it would be cleaner semantically to copy the \cs_set_eq:NN
\bool_set_eq:cN family of functions, we copy \tl_set_eq:NN because that has the correct checking code.
\bool_set_eq:Nc
\bool_set_eq:cc
8388 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
8389 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
\bool_gset_eq:NN
\bool_gset_eq:cN
8390 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
8391 \cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
\bool_gset_eq:Nc
\bool_gset_eq:cc

```

(End of definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 68.)

```

\bool_set:Nn This function evaluates a boolean expression and assigns the first argument the meaning
\bool_set:cn \c_true_bool or \c_false_bool. Again, we include some checking code. It is important
\bool_gset:Nn to evaluate the expression before applying the \chardef primitive, because that primitive
\bool_gset:cn sets the left-hand side to \scan_stop: before looking for the right-hand side.

```

```

8392 \cs_new_protected:Npn \bool_set:Nn #1#2
8393 {
8394   \exp_last_unbraced:NNNf
8395   \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8396 }
8397 \cs_new_protected:Npn \bool_gset:Nn #1#2
8398 {
8399   \exp_last_unbraced:NNNNf
8400   \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8401 }
8402 \cs_generate_variant:Nn \bool_set:Nn { c }
8403 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End of definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 68.)

```

\bool_set_inverse:N Set to false or true locally or globally.
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c
8404 \cs_new_protected:Npn \bool_set_inverse:N #1
8405 { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
8406 \cs_generate_variant:Nn \bool_set_inverse:N { c }
8407 \cs_new_protected:Npn \bool_gset_inverse:N #1
8408 { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
8409 \cs_generate_variant:Nn \bool_gset_inverse:N { c }

```

(End of definition for `\bool_set_inverse:N` and `\bool_gset_inverse:N`. These functions are documented on page 69.)

51.4 Internal auxiliaries

`\q__bool_recursion_tail` Internal recursion quarks.

```
\q__bool_recursion_stop 8410 \quark_new:N \q__bool_recursion_tail
                        8411 \quark_new:N \q__bool_recursion_stop
```

(End of definition for `\q__bool_recursion_tail` and `\q__bool_recursion_stop`.)

`\__bool_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

```
8412 \cs_new:Npn \__bool_use_i_delimit_by_q_recursion_stop:nw
8413   #1 #2 \q__bool_recursion_stop {#1}
```

(End of definition for `\__bool_use_i_delimit_by_q_recursion_stop:nw`.)

`\__bool_if_recursion_tail_stop_do:nn` Functions to query recursion quarks.

```
8414 \__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn
```

(End of definition for `\__bool_if_recursion_tail_stop_do:nn`.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\bool_if_p:c
\bool_if:NTF 8415 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:cTF 8416 {
                8417   \if_bool:N #1
                8418     \prg_return_true:
                8419   \else:
                8420     \prg_return_false:
                8421   \fi:
                8422 }
8423 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }
```

(End of definition for `\bool_if:N`. This function is documented on page 69.)

`\bool_to_str:N` Expands to string literal true or false.

```
\bool_to_str:c 8424 \cs_new:Npe \bool_to_str:N #1
\bool_to_str:n 8425 {
                8426   \exp_not:N \bool_if:N {TF} #1
                8427   { \tl_to_str:n { true } } { \tl_to_str:n { false } }
                8428 }
                8429 \cs_generate_variant:Nn \bool_to_str:N { c }
                8430 \cs_new:Npe \bool_to_str:n #1
                8431 {
                8432   \exp_not:N \bool_if:n {TF} {#1}
                8433   { \tl_to_str:n { true } } { \tl_to_str:n { false } }
                8434 }
```

(End of definition for `\bool_to_str:N` and `\bool_to_str:n`. These functions are documented on page 69.)

\bool_show:n Show the truth value of the boolean.

```
\bool_log:n      8435 \cs_new_protected:Npn \bool_show:n
                  8436 { \__kernel_msg_show_eval:Nn \bool_to_str:n }
                  8437 \cs_new_protected:Npn \bool_log:n
                  8438 { \__kernel_msg_log_eval:Nn \bool_to_str:n }
```

(End of definition for \bool_show:n and \bool_log:n. These functions are documented on page 69.)

\bool_show:N Show the truth value of the boolean, as true or false.

```
\bool_show:c      8439 \cs_new_protected:Npn \bool_show:N { \__bool_show:NN \tl_show:n }
\bool_log:N      8440 \cs_generate_variant:Nn \bool_show:N { c }
\bool_log:c      8441 \cs_new_protected:Npn \bool_log:N { \__bool_show:NN \tl_log:n }
\__bool_show:NN  8442 \cs_generate_variant:Nn \bool_log:N { c }
                  8443 \cs_new_protected:Npn \__bool_show:NN #1#2
                  8444 {
                  8445     \__kernel_chk_defined:NT #2
                  8446     {
                  8447         \token_case_meaning:NnF #2
                  8448         {
                  8449             \c_true_bool { \exp_args:Ne #1 { \token_to_str:N #2 = true } }
                  8450             \c_false_bool { \exp_args:Ne #1 { \token_to_str:N #2 = false } }
                  8451         }
                  8452         {
                  8453             \msg_error:nneee { kernel } { bad-type }
                  8454             { \token_to_str:N #2 } { \token_to_meaning:N #2 } { bool }
                  8455         }
                  8456     }
                  8457 }
```

(End of definition for \bool_show:N, \bool_log:N, and __bool_show:NN. These functions are documented on page 69.)

\l_tmpa_bool A few booleans just if you need them.

```
\l_tmpb_bool      8458 \bool_new:N \l_tmpa_bool
\g_tmpa_bool      8459 \bool_new:N \l_tmpb_bool
\g_tmpb_bool      8460 \bool_new:N \g_tmpa_bool
                  8461 \bool_new:N \g_tmpb_bool
```

(End of definition for \l_tmpa_bool and others. These variables are documented on page 69.)

\bool_if_exist_p:N Copies of the cs functions defined in l3basics.

```
\bool_if_exist_p:c  8462 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:N  8463 { TF , T , F , p }
\bool_if_exist:c  8464 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
\bool_if_exist:c  8465 { TF , T , F , p }
```

(End of definition for \bool_if_exist:N. This function is documented on page 69.)

51.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with (and) for grouping, ! for logical “Not”, && for logical “And” and || for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

`<true>`**And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

`<false>`**And** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return `<false>`.

`<true>`**Or** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return `<true>`.

`<false>`**Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

`<true>`**Close** Current truth value is true, Close seen, return `<true>`.

`<false>`**Close** Current truth value is false, Close seen, return `<false>`.

```

8466 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
8467 {
8468   \if_predicate:w \bool_if_p:n {#1}
8469   \prg_return_true:
8470   \else:
8471     \prg_return_false:
8472   \fi:
8473 }
```

(End of definition for `\bool_if:nTF`. This function is documented on page 71.)

`\bool_if_p:n` To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty #1 is an error. The auxiliary `__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space, then returns `\c_true_bool` or `\c_false_bool` as appropriate. This extra work around is because in a `\bool_set:Nn`, the underlying `\chardef` turns

the bool being set temporarily equal to `\relax`, thus assigning a boolean to itself would fail (gh/1055). For the general case, first issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for T_EX. This group is closed after `\__bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

8474 \cs_new:Npn \bool_if_p:n { \exp_args:Nf \__bool_if_p:n }
8475 \cs_new:Npn \__bool_if_p:n #1
8476 {
8477   \tl_if_empty:oT { \use_none:nn #1 . } { \__bool_if_p_aux:w }
8478   \group_align_safe_begin:
8479   \exp_after:wN
8480   \group_align_safe_end:
8481   \exp:w \exp_end_continue_f:w % (
8482   \__bool_get_next:NN \use_i:nnnn #1 )
8483 }
8484 \cs_new:Npn \__bool_if_p_aux:w #1 \use_i:nnnn #2#3
8485 { \bool_if:NTF #2 \c_true_bool \c_false_bool }

```

(End of definition for `\bool_if_p:n`, `\__bool_if_p:n`, and `\__bool_if_p_aux:w`. This function is documented on page 71.)

`\__bool_get_next:NN` The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_ii:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression which follows until the next unmatched closing parenthesis. For instance “`\__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool )`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a `!`) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering `||`, not only parentheses. This code itself is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate.

```

8486 \cs_new:Npn \__bool_get_next:NN #1#2
8487 {
8488   \use:c
8489   {
8490     __bool_
8491     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
8492     :Nw
8493   }
8494   #1 #2
8495 }

```

(End of definition for `\__bool_get_next:NN`.)

`\__bool_!:Nw` The Not operation reverses the logic: it discards the `!` token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after `\c_true_bool||` or `\c_false_bool&&` the `!` is ignored.

```

8496 \cs_new:cpn { __bool_!:Nw } #1#2
8497 {
8498   \exp_after:wN \__bool_get_next:NN
8499   #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
8500 }

```

(End of definition for _bool_!:Nw.)

bool(:Nw The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```

8501 \cs_new:cpn { \_bool\_(:Nw } #1#2
8502   {
8503     \exp_after:wN \_bool_choose:NNN \exp_after:wN #1
8504     \int_value:w \_bool_get_next:NN \use_i:nnnn
8505   }

```

(End of definition for _bool_(:Nw.)

_bool_p:Nw If what follows GetNext is neither ! nor (, evaluate the predicate using the primitive \int_value:w. The canonical true and false values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```

8506 \cs_new:cpn { \_bool\_p:Nw } #1
8507   { \exp_after:wN \_bool_choose:NNN \exp_after:wN #1 \int_value:w }

```

(End of definition for _bool_p:Nw.)

_bool_choose:NNN The arguments are #1: a function such as \use_i:nnnn, #2: 0 or 1 encoding the current truth value, #3: the next operation, And, Or or Close. We distinguish three cases according to a combination of #1 and #2. Case 2 is when #1 is \use_iii:nnnn (state 3), namely after \c_true_bool ||. Case 1 is when #1 is \use_i:nnnn and #2 is true or when #1 is \use_ii:nnnn and #2 is false, for instance for !\c_false_bool. Case 0 includes the same with true/false interchanged and the case where #1 is \use_iv:nnnn namely after \c_false_bool &&.

bool|_0: When seeing) the current subexpression is done, leave the appropriate boolean.
bool|_1: When seeing & in case 0 go into state 4, equivalent to having seen \c_false_bool &&.
bool|_2: In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \c_false_bool && because that binds more tightly than ||. In the other two cases start skipping for \c_true_bool ||.

```

8508 \cs_new:Npn \_bool_choose:NNN #1#2#3
8509   {
8510     \use:c
8511     {
8512       \_bool\_ \token_to_str:N #3 _
8513       #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
8514     }
8515   }
8516 \cs_new:cpn { \_bool\_|_0: } { \c_false_bool }
8517 \cs_new:cpn { \_bool\_|_1: } { \c_true_bool }
8518 \cs_new:cpn { \_bool\_|_2: } { \c_true_bool }
8519 \cs_new:cpn { \_bool\_&_0: } & { \_bool_get_next:NN \use_iv:nnnn }
8520 \cs_new:cpn { \_bool\_&_1: } & { \_bool_get_next:NN \use_i:nnnn }
8521 \cs_new:cpn { \_bool\_&_2: } & { \_bool_get_next:NN \use_iii:nnnn }
8522 \cs_new:cpn { \_bool\_|_0: } | { \_bool_get_next:NN \use_i:nnnn }
8523 \cs_new:cpn { \_bool\_|_1: } | { \_bool_get_next:NN \use_iii:nnnn }
8524 \cs_new:cpn { \_bool\_|_2: } | { \_bool_get_next:NN \use_iii:nnnn }

```

(End of definition for `\_bool_choose:NNN` and others.)

`\bool_lazy_all_p:n` Go through the list of expressions, stopping whenever an expression is `false`. If the end is reached without finding any `false` expression, then the result is `true`.

`\bool_lazy_all:nTF`
`\_bool_lazy_all:n`

```

8525 \cs_new:Npn \bool_lazy_all_p:n #1
8526 { \_bool_lazy_all:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
8527 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
8528 {
8529   \if_predicate:w \bool_lazy_all_p:n {#1}
8530   \prg_return_true:
8531   \else:
8532     \prg_return_false:
8533   \fi:
8534 }
8535 \cs_new:Npn \_bool_lazy_all:n #1
8536 {
8537   \_bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
8538   \bool_if:nF {#1}
8539   { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
8540   \_bool_lazy_all:n
8541 }
```

(End of definition for `\bool_lazy_all:nTF` and `\_bool_lazy_all:n`. This function is documented on page 71.)

`\bool_lazy_and_p:nn` Only evaluate the second expression if the first is `true`. Note that `#2` must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since `#2` may contain unbalanced `TeX` conditionals.

`\bool_lazy_and:nnTF`

```

8542 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
8543 {
8544   \if_predicate:w
8545     \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
8546   \prg_return_true:
8547   \else:
8548     \prg_return_false:
8549   \fi:
8550 }
```

(End of definition for `\bool_lazy_and:nnTF`. This function is documented on page 71.)

`\bool_lazy_any_p:n` Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

`\bool_lazy_any:nTF`
`\_bool_lazy_any:n`

```

8551 \cs_new:Npn \bool_lazy_any_p:n #1
8552 { \_bool_lazy_any:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
8553 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
8554 {
8555   \if_predicate:w \bool_lazy_any_p:n {#1}
8556   \prg_return_true:
8557   \else:
8558     \prg_return_false:
8559   \fi:
8560 }
8561 \cs_new:Npn \_bool_lazy_any:n #1
8562 {
```

```

8563     \_bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
8564     \bool_if:nT {#1}
8565         { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
8566     \_bool_lazy_any:n
8567 }

```

(End of definition for \bool_lazy_any:nTF and _bool_lazy_any:n. This function is documented on page 71.)

\bool_lazy_or_p:nn Only evaluate the second expression if the first is false.

```

\bool_lazy_or:nnTF
8568 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
8569 {
8570     \if_predicate:w
8571         \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
8572     \prg_return_true:
8573     \else:
8574         \prg_return_false:
8575     \fi:
8576 }

```

(End of definition for \bool_lazy_or:nnTF. This function is documented on page 71.)

\bool_not_p:n The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```

8577 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }

```

(End of definition for \bool_not_p:n. This function is documented on page 71.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```

\bool_xor:nnTF
8578 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
8579 {
8580     \bool_if:nT {#1} \reverse_if:N
8581     \if_predicate:w \bool_if_p:n {#2}
8582     \prg_return_true:
8583     \else:
8584     \prg_return_false:
8585     \fi:
8586 }

```

(End of definition for \bool_xor:nnTF. This function is documented on page 72.)

51.6 Logical loops

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

```

\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn
8587 \cs_new:Npn \bool_while_do:Nn #1#2
8588 { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
8589 \cs_new:Npn \bool_until_do:Nn #1#2
8590 { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
8591 \cs_generate_variant:Nn \bool_while_do:Nn { c }
8592 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End of definition for \bool_while_do:Nn and \bool_until_do:Nn. These functions are documented on page 72.)

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested
 \bool_do_while:cn after executing the body. Otherwise identical to the above functions.
 \bool_do_until:Nn
 \bool_do_until:cn

```

8593 \cs_new:Npn \bool_do_while:Nn #1#2
8594   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
8595 \cs_new:Npn \bool_do_until:Nn #1#2
8596   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
8597 \cs_generate_variant:Nn \bool_do_while:Nn { c }
8598 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End of definition for \bool_do_while:Nn and \bool_do_until:Nn. These functions are documented on page 72.)

\bool_while_do:nn Loop functions with the test either before or after the first body expansion.

```

8599 \cs_new:Npn \bool_while_do:nn #1#2
8600   {
8601     \bool_if:nT {#1}
8602     {
8603       #2
8604       \bool_while_do:nn {#1} {#2}
8605     }
8606   }
8607 \cs_new:Npn \bool_do_while:nn #1#2
8608   {
8609     #2
8610     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
8611   }
8612 \cs_new:Npn \bool_until_do:nn #1#2
8613   {
8614     \bool_if:nF {#1}
8615     {
8616       #2
8617       \bool_until_do:nn {#1} {#2}
8618     }
8619   }
8620 \cs_new:Npn \bool_do_until:nn #1#2
8621   {
8622     #2
8623     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
8624   }

```

(End of definition for \bool_while_do:nn and others. These functions are documented on page 73.)

\s__bool_mark Internal scan marks.

```

8625 \scan_new:N \s__bool_mark
8626 \scan_new:N \s__bool_stop

```

(End of definition for \s__bool_mark and \s__bool_stop.)

\bool_case:n For boolean cases the overall idea is the same as for \str_case:nnTF as described in l3str.

\bool_case:nTF

```

8627 \cs_new:Npn \bool_case:nTF
8628   { \exp:w \__bool_case:nTF
__bool_case:NnTF
__bool_case:w
a\__bool_case_end:nw

```

```

8629 \cs_new:Npn \bool_case:nT #1#2
8630 { \exp:w \__bool_case:nTF {#1} {#2} { } }
8631 \cs_new:Npn \bool_case:nF #1
8632 { \exp:w \__bool_case:nTF {#1} { } }
8633 \cs_new:Npn \bool_case:n #1
8634 { \exp:w \__bool_case:nTF {#1} { } { } }
8635 \cs_new:Npn \__bool_case:nTF #1#2#3
8636 {
8637   \__bool_case:w
8638   #1 \c_true_bool { } \s__bool_mark {#2} \s__bool_mark {#3} \s__bool_stop
8639 }
8640 \cs_new:Npn \__bool_case:w #1#2
8641 {
8642   \bool_if:nTF {#1}
8643     { \__bool_case_end:nw {#2} }
8644     { \__bool_case:w }
8645 }
8646 \cs_new:Npn \__bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
8647 { \exp_end: #1 #4 }

```

(End of definition for `\bool_case:nTF` and others. This function is documented on page 73.)

51.7 Producing multiple copies

```

8648 <@@=prg>

```

\prg_replicate:nn

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {<code>} }`. An alternative approach is to create a string of m's with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

\__prg_replicate:N
\__prg_replicate_first:N
\__prg_replicate_
\__prg_replicate_0:n
\__prg_replicate_1:n
\__prg_replicate_2:n
\__prg_replicate_3:n
\__prg_replicate_4:n
\__prg_replicate_5:n
\__prg_replicate_6:n
\__prg_replicate_7:n
\__prg_replicate_8:n
\__prg_replicate_9:n
\__prg_replicate_first_0:n
\__prg_replicate_first_1:n
\__prg_replicate_first_2:n
\__prg_replicate_first_3:n
\__prg_replicate_first_4:n
\__prg_replicate_first_5:n
\__prg_replicate_first_6:n
\__prg_replicate_first_7:n
\__prg_replicate_first_8:n
\__prg_replicate_first_9:n

```

```

8649 \cs_new:Npn \prg_replicate:nn #1
8650 {
8651   \exp:w
8652     \exp_after:wN \__prg_replicate_first:N
8653     \int_value:w \int_eval:n {#1}
8654     \cs_end:

```


conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
8697 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
8698   { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_vertical:TF`. This function is documented on page 74.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```
\mode_if_horizontal:TF 8699 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
8700   { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_horizontal:TF`. This function is documented on page 73.)

`\mode_if_inner_p:` For testing inner mode.

```
\mode_if_inner:TF 8701 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
8702   { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_inner:TF`. This function is documented on page 74.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
8703 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
8704   { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End of definition for `\mode_if_math:TF`. This function is documented on page 74.)

51.9 Internal programming functions

`\group_align_safe_begin:` `\group_align_safe_end:` T_EX’s alignment structures present many problems. As Knuth says himself in *T_EX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issue a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T_EX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using behavior documented only in Appendix D of *The T_EXbook*... In short evaluating ‘{ and ‘} as numbers will not change the counter T_EX uses to keep track of its state in an alignment, whereas gobbling a brace using `\if_false:` will affect T_EX’s state without producing any real group. We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
8705 \group_begin:
8706 \tex_catcode:D ‘\^^@ = 2 \exp_stop_f:
8707 \cs_new:Npn \group_align_safe_begin:
8708   { \exp:w \if_false: { \fi: ‘^^@ \exp_stop_f: }
8709 \tex_catcode:D ‘\^^@ = 1 \exp_stop_f:
8710 \cs_new:Npn \group_align_safe_end:
8711   { \exp:w ‘^^@ \if_false: } \fi: \exp_stop_f: }
8712 \group_end:
```


(End of definition for \group_align_safe_begin: and \group_align_safe_end:. These functions are documented on page 75.)

`\g__kernel_prg_map_int` A nesting counter for mapping.

8713 `\int_new:N \g__kernel_prg_map_int`

(End of definition for \g__kernel_prg_map_int.)

`\prg_break_point:Nn` These are defined in l3basics, as they are needed “early”. This is just a reminder that is the case!
`\prg_map_break:Nn`

(End of definition for \prg_break_point:Nn and \prg_map_break:Nn. These functions are documented on page 74.)

`\prg_break_point:` Also done in l3basics.

`\prg_break:`

`\prg_break:n`

(End of definition for \prg_break_point:, \prg_break:, and \prg_break:n. These functions are documented on page 75.)

8714 `</code>`

Chapter 52

l3sys implementation

8715 <@@=sys>

52.1 Kernel code

8716 <*code>

\l__sys_tmp_tl

8717 \tl_new:N \l__sys_tmp_tl

(End of definition for \l__sys_tmp_tl.)

52.1.1 Detecting the engine

__sys_const:nn Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
8718 \cs_new_protected:Npn \__sys_const:nn #1#2
8719 {
8720   \bool_if:nTF {#2}
8721   {
8722     \cs_new_eq:cN { #1 :T } \use:n
8723     \cs_new_eq:cN { #1 :F } \use_none:n
8724     \cs_new_eq:cN { #1 :TF } \use_i:nn
8725     \cs_new_eq:cN { #1 _p: } \c_true_bool
8726   }
8727   {
8728     \cs_new_eq:cN { #1 :T } \use_none:n
8729     \cs_new_eq:cN { #1 :F } \use:n
8730     \cs_new_eq:cN { #1 :TF } \use_ii:nn
8731     \cs_new_eq:cN { #1 _p: } \c_false_bool
8732   }
8733 }
```

(End of definition for __sys_const:nn.)

\sys_if_engine luatex_p: Set up the engine tests on the basis exactly one test should be true. Mainly a case of
\sys_if_engine luatex:TF looking for the appropriate marker primitive.

\sys_if_engine pdftex_p: 8734 \str_const:Ne \c_sys_engine_str
\sys_if_engine pdftex:TF 8735 {

\sys_if_engine ptex_p:

\sys_if_engine ptex:TF

\sys_if_engine uptex_p:

\sys_if_engine uptex:TF

\sys_if_engine xetex_p:

\sys_if_engine xetex:TF

\c_sys_engine_str

```

8736 \cs_if_exist:NT \tex luatexversion:D { luatex }
8737 \cs_if_exist:NT \tex pdftexversion:D { pdftex }
8738 \cs_if_exist:NT \tex kanjiskip:D
8739 {
8740   \cs_if_exist:NTF \tex enablecjktoken:D
8741     { uptex }
8742     { ptex }
8743   }
8744 \cs_if_exist:NT \tex XeTeXversion:D { xetex }
8745 }
8746 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
8747 {
8748   \__sys_const:nn { sys_if_engine_ #1 }
8749   { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
8750 }

```

(End of definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 77.)

```

\sys_if_engine_opentype_p: ★
\sys_if_engine_opentype:TF ★

```

```

8751 \__sys_const:nn
8752 { sys_if_engine_opentype }
8753 { \cs_if_exist_p:N \tex_Umathcode:D }

```

`\c_sys_engine_exec_str`
`\c_sys_engine_format_str`

Take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a “filtered” version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because L^AT_EX uses the LuaH^BT_EX engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdflatex` from `latex` (which is pdf_TE_X in DVI mode). This differentiation, however, is reliable only if the user doesn’t change `\tex_pdfoutput:D` before loading this code.

```

8754 \group_begin:
8755   \cs_set_eq:NN \lua_now:e \tex_directlua:D
8756   \str_const:Nc \c_sys_engine_exec_str
8757   {
8758     \sys_if_engine_pdftex:T { pdf }
8759     \sys_if_engine_xetex:T { xe }
8760     \sys_if_engine_ptex:T { ep }
8761     \sys_if_engine_uptex:T { eup }
8762     \sys_if_engine_luatex:T
8763     {
8764       lua \lua_now:e
8765       {
8766         if (pcall(require, 'luaharfbuzz')) then ~
8767           tex.print("hb") ~
8768         end
8769       }
8770     }
8771   tex
8772 }

```

```

8773 \group_end:
8774 \str_const:Ne \c_sys_engine_format_str
8775 {
8776   \cs_if_exist:NTF \fmtname
8777   {
8778     \bool_lazy_or:nnTF
8779     { \str_if_eq_p:Vn \fmtname { plain } }
8780     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
8781     {
8782       \sys_if_engine_pdftex:T
8783       { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
8784       \sys_if_engine_xetex:T { xe }
8785       \sys_if_engine_ptex:T { p }
8786       \sys_if_engine_uptex:T { up }
8787       \sys_if_engine luatex:T
8788       {
8789         \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
8790         lua
8791       }
8792       \str_if_eq:VnTF \fmtname { LaTeX2e }
8793       { latex }
8794       {
8795         \bool_lazy_and:nnT
8796         { \sys_if_engine_pdftex_p: }
8797         { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
8798         { e }
8799         tex
8800       }
8801     }
8802     { \fmtname }
8803   }
8804   { unknown }
8805 }

```

(End of definition for `\c_sys_engine_exec_str` and `\c_sys_engine_format_str`. These variables are documented on page 77.)

`\c_sys_engine_version_str` Various different engines, various different ways to extract the data!

```

8806 \str_const:Ne \c_sys_engine_version_str
8807 {
8808   \str_case:on \c_sys_engine_str
8809   {
8810     { pdftex }
8811     {
8812       \int_div_truncate:nn { \tex_pdfptextversion:D } { 100 }
8813       .
8814       \int_mod:nn { \tex_pdfptextversion:D } { 100 }
8815       .
8816       \tex_pdfptextrevision:D
8817     }
8818     { ptex }
8819     {
8820       \cs_if_exist:NT \tex_ptextversion:D
8821       {

```

```

8822         p
8823         \int_use:N \tex_ptexversion:D
8824         .
8825         \int_use:N \tex_ptexminorversion:D
8826         \tex_ptexrevision:D
8827         -
8828         \int_use:N \tex_epTeXversion:D
8829     }
8830 }
8831 { luatex }
8832 {
8833     \int_div_truncate:nn { \tex_luatexversion:D } { 100 }
8834     .
8835     \int_mod:nn { \tex_luatexversion:D } { 100 }
8836     .
8837     \tex_luatexrevision:D
8838 }
8839 { uptex }
8840 {
8841     \cs_if_exist:NT \tex_ptexversion:D
8842     {
8843         p
8844         \int_use:N \tex_ptexversion:D
8845         .
8846         \int_use:N \tex_ptexminorversion:D
8847         \tex_ptexrevision:D
8848         -
8849         u
8850         \int_use:N \tex_uptexversion:D
8851         \tex_uptexrevision:D
8852         -
8853         \int_use:N \tex_epTeXversion:D
8854     }
8855 }
8856 { xetex }
8857 {
8858     \int_use:N \tex_XeTeXversion:D
8859     \tex_XeTeXrevision:D
8860 }
8861 }
8862 }

```

(End of definition for `\c_sys_engine_version_str`. This variable is documented on page 78.)

52.1.2 Platform

`\sys_if_platform_unix_p:` Setting these up requires the file module (file lookup), so is actually implemented there.

`\sys_if_platform_unix:TF`

`\sys_if_platform_windows_p:`

`\sys_if_platform_windows:TF`

`\c_sys_platform_str`

(End of definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform_str`. These functions are documented on page 79.)

52.1.3 Configurations

Loading the backend code is pretty simply: check that the backend is valid, then load it up.

`\sys_load_backend:n`

`\__sys_load_backend_check:N`

`\c_sys_backend_str`

```

8863 \cs_new_protected:Npn \sys_load_backend:n #1
8864 {
8865   \sys_finalize:
8866   \str_if_exist:NTF \c_sys_backend_str
8867   {
8868     \str_if_eq:VnF \c_sys_backend_str {#1}
8869     { \msg_error:nn { sys } { backend-set } }
8870   }
8871   {
8872     \tl_if_blank:nF {#1}
8873     { \tl_gset:Nn \g__sys_backend_tl {#1} }
8874     \__sys_load_backend_check:N \g__sys_backend_tl
8875     \str_const:Ne \c_sys_backend_str { \g__sys_backend_tl }
8876     \__kernel_sys_configuration_load:n
8877     { l3backend- \c_sys_backend_str }
8878   }
8879 }
8880 \cs_new_protected:Npn \__sys_load_backend_check:N #1
8881 {
8882   \sys_if_engine_xetex:TF
8883   {
8884     \str_case:VnF #1
8885     {
8886       { dvisvgm } { }
8887       { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
8888       { xetex } { }
8889     }
8890     {
8891       \msg_error:nnee { sys } { wrong-backend }
8892       #1 { xetex }
8893       \tl_gset:Nn #1 { xetex }
8894     }
8895   }
8896   {
8897     \sys_if_output_pdf:TF
8898     {
8899       \str_if_eq:VnTF #1 { pdfmode }
8900       {
8901         \sys_if_engine luatex:TF
8902         { \tl_gset:Nn #1 { luatex } }
8903         { \tl_gset:Nn #1 { pdftex } }
8904       }
8905       {
8906         \bool_lazy_or:nnF
8907         { \str_if_eq_p:Vn #1 { luatex } }
8908         { \str_if_eq_p:Vn #1 { pdftex } }
8909         {
8910           \msg_error:nnee { sys } { wrong-backend }
8911           #1 { \sys_if_engine luatex:TF { luatex } { pdftex } }
8912           \sys_if_engine luatex:TF

```

```

8913             { \tl_gset:Nn #1 { luatex } }
8914             { \tl_gset:Nn #1 { pdftex } }
8915         }
8916     }
8917 }
8918 {
8919     \str_case:NnF #1
8920     {
8921         { dvipdfmx } { }
8922         { dvips } { }
8923         { dvisvgm } { }
8924     }
8925     {
8926         \msg_error:nnee { sys } { wrong-backend }
8927         #1 { dvips }
8928         \tl_gset:Nn #1 { dvips }
8929     }
8930 }
8931 }
8932 }

```

(End of definition for `\sys_load_backend:n`, `__sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page 81.)

`\sys_ensure_backend:` A simple wrapper.

```

8933 \cs_new_protected:Npn \sys_ensure_backend:
8934 {
8935     \str_if_exist:NF \c_sys_backend_str
8936     { \sys_load_backend:n { } }
8937 }

```

(End of definition for `\sys_ensure_backend:.` This function is documented on page 81.)

`\g__sys_debug_bool`

```

8938 \bool_new:N \g__sys_debug_bool

```

(End of definition for `\g__sys_debug_bool`.)

`\sys_load_debug:` The most complicated thing here is that we can only use `__kernel_sys_configuration_load:n` in the preamble in L^AT_EX.

```

8939 \cs_new_protected:Npn \sys_load_debug:
8940 {
8941     \bool_if:NF \g__sys_debug_bool
8942     { \__kernel_sys_configuration_load:n { l3debug } }
8943     \bool_gset_true:N \g__sys_debug_bool
8944 }
8945 \cs_if_exist:NT \@expl@finalise@setup@@
8946 {
8947     \tl_gput_right:Nn \@expl@finalise@setup@@
8948     {
8949         \tl_gput_right:Nn \@kernel@after@begindocument
8950         {
8951             \cs_gset_protected:Npn \sys_load_debug:
8952             { \msg_error:nn { sys } { load-debug-in-preamble } }
8953         }
8954     }
8955 }

```

```

8954     }
8955 }

```

(End of definition for `\sys_load_debug`:. This function is documented on page 82.)

52.1.4 Access to the shell

`\c__sys_marker_tl` The same idea as the marker for rescanning token lists.

```

8956 \tl_const:Ne \c__sys_marker_tl { : \token_to_str:N : }

```

(End of definition for `\c__sys_marker_tl`.)

`\sys_get_shell:nnNTF` Setting using a shell is at this level just a slightly specialized file operation, with an additional check for quotes, as these are not supported.

```

\sys_get_shell:nnN
  \__sys_get:nnN
  \__sys_get_do:Nw
8957 \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
8958 {
8959   \sys_get_shell:nnNF {#1} {#2} #3
8960   { \tl_set:Nn #3 { \q_no_value } }
8961 }
8962 \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
8963 {
8964   \sys_if_shell:TF
8965   { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
8966   { \prg_return_false: }
8967 }
8968 \cs_new_protected:Npn \__sys_get:nnN #1#2#3
8969 {
8970   \tl_if_in:nnTF {#1} { " }
8971   {
8972     \msg_error:nne
8973     { kernel } { quote-in-shell } {#1}
8974     \prg_return_false:
8975   }
8976   {
8977     \group_begin:
8978     \if_false: { \fi:
8979       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
8980       \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
8981       #2 \scan_stop:
8982       \exp_after:wN \__sys_get_do:Nw
8983       \exp_after:wN #3
8984       \exp_after:wN \prg_do_nothing:
8985       \tex_input:D | "#1" \scan_stop:
8986     \if_false: } \fi:
8987     \prg_return_true:
8988   }
8989 }
8990 \exp_args:Nno \use:nn
8991 { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
8992 { \c__sys_marker_tl }
8993 {
8994   \group_end:
8995   \tl_set:No #1 {#2}
8996 }

```


(End of definition for `\sys_get_shell:nnNTF` and others. These functions are documented on page 79.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```
8997 \sys_if_engine luatex:F
8998   { \int_const:Nn \c__sys_shell_stream_int { 18 } }
8999 </code>
```

(End of definition for `\c__sys_shell_stream_int`.)

```
\sys_shell_now:n Execute commands through shell escape immediately.
\sys_shell_now:e   For LuaTeX, we use a pseudo-primitive to do the actual work.
\sys_shell_now:x
\__sys_shell_now:e
9000 (*lua)
9001 do
9002   local os_exec = os.execute
9003
9004   local function shellescape(cmd)
9005     local status,msg = os_exec(cmd)
9006     if status == nil then
9007       write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ").\n")
9008     elseif status == 0 then
9009       write_nl("log","runsystem(" .. cmd .. ")...executed.\n")
9010     else
9011       write_nl("log","runsystem(" .. cmd .. ")...failed. " .. (msg or "") .. "\n")
9012     end
9013   end
9014   luacmd("__sys_shell_now:e", function()
9015     shellescape(scan_string())
9016   end, "global", "protected")
9017 </lua>
9018 (*code)
9019 \sys_if_engine luatex:TF
9020   {
9021     \cs_new_protected:Npn \sys_shell_now:n #1
9022       { \__sys_shell_now:e { \exp_not:n {#1} } }
9023   }
9024   {
9025     \cs_new_protected:Npn \sys_shell_now:n #1
9026       { \iow_now:Nn \c__sys_shell_stream_int {#1} }
9027   }
9028 \cs_generate_variant:Nn \sys_shell_now:n { e, x }
9029 </code>
```

(End of definition for `\sys_shell_now:n` and `\__sys_shell_now:e`. This function is documented on page 80.)

```
\sys_shell_shipout:n Execute commands through shell escape at shipout.
\sys_shell_shipout:e   For LuaTeX, we use the same helper as above but delayed using a late_lua whatsit.
\sys_shell_shipout:x   Creating a late_lua whatsit works a bit different if we are running under ConTeXt.
\__sys_shell_shipout:e
9030 (*lua)
9031   local new_latelua = nodes and nodes.nuts and nodes.nuts.pool and nodes.nuts.pool.latelua
9032   local whatsit_id = node.id'whatsit'
9033   local latelua_sub = node.subtype'late_lua'
9034   local node_new = node.direct.new
9035   local setfield = node.direct.setwhatsitfield or node.direct.setfield
```

```

9036     return function(f)
9037         local n = node_new(whatsit_id, latelua_sub)
9038         setfield(n, 'data', f)
9039         return n
9040     end
9041 end)()
9042 local node_write = node.direct.write
9043
9044 luacmd("__sys_shell_shipout:e", function()
9045     local cmd = scan_string()
9046     node_write(new_latelua(function() shellescape(cmd) end))
9047 end, "global", "protected")
9048 end
9049  $\langle$ /lua $\rangle$ 
9050  $\langle$ *code $\rangle$ 
9051 \sys_if_engine luatex:TF
9052 {
9053     \cs_new_protected:Npn \sys_shell_shipout:n #1
9054     { \__sys_shell_shipout:e { \exp_not:n {#1} } }
9055 }
9056 {
9057     \cs_new_protected:Npn \sys_shell_shipout:n #1
9058     { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
9059 }
9060 \cs_generate_variant:Nn \sys_shell_shipout:n { e , x }

```

(End of definition for `\sys_shell_shipout:n` and `\__sys_shell_shipout:e`. This function is documented on page 80.)

52.2 Dynamic (every job) code

```

\__kernel_sys_everyjob:
  \__sys_everyjob:n
  \g__sys_everyjob_tl
9061 \cs_new_protected:Npn \__kernel_sys_everyjob:
9062 {
9063     \tl_use:N \g__sys_everyjob_tl
9064     \tl_gclear:N \g__sys_everyjob_tl
9065 }
9066 \cs_new_protected:Npn \__sys_everyjob:n #1
9067 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
9068 \tl_new:N \g__sys_everyjob_tl

```

(End of definition for `\__kernel_sys_everyjob:`, `\__sys_everyjob:n`, and `\g__sys_everyjob_tl`.)

52.2.1 The name of the job

`\c_sys_jobname_str` Inherited from the L^AT_EX3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed from `texput`, and thus we would have the wrong result.

```

9069 \__sys_everyjob:n
9070 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }

```

(End of definition for `\c_sys_jobname_str`. This variable is documented on page 76.)

52.2.2 Time and date

`\c_sys_minute_int` `\c_sys_hour_int` `\c_sys_day_int` `\c_sys_month_int` `\c_sys_year_int` Copies of the information provided by T_EX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have `\primitive` and it is working correctly. For IniT_EX of course that is all redundant but does no harm.

```

9071 \__sys_everyjob:n
9072 {
9073   \group_begin:
9074   \cs_set:Npn \__sys_tmp:w #1
9075   {
9076     \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
9077     { #1 }
9078     {
9079       \cs_if_exist:NTF \tex_primitive:D
9080       {
9081         \bool_lazy_and:nnTF
9082         { \sys_if_engine_xetex_p: }
9083         {
9084           \int_compare_p:nNn
9085           { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
9086           < { 99999 }
9087         }
9088         { 0 }
9089         { \tex_primitive:D #1 }
9090       }
9091       { 0 }
9092     }
9093   }
9094   \int_const:Nn \c_sys_minute_int
9095   { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
9096   \int_const:Nn \c_sys_hour_int
9097   { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
9098   \int_const:Nn \c_sys_day_int { \__sys_tmp:w \day }
9099   \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
9100   \int_const:Nn \c_sys_year_int { \__sys_tmp:w \year }
9101   \group_end:
9102 }

```

(End of definition for `\c_sys_minute_int` and others. These variables are documented on page 76.)

`\c_sys_timestamp_str` A simple expansion: LuaT_EX chokes if we use `\pdffeedback` here, hence the direct use of Lua. Notice that the function there is in the pdf library but isn't actually tied to PDF.

```

9103 \__sys_everyjob:n
9104 {
9105   \str_const:Ne \c_sys_timestamp_str
9106   {
9107     \cs_if_exist:NTF \tex_directlua:D
9108     { \tex_directlua:D { tex.print(pdf.getcreationdate()) } }
9109     { \tex_creationdate:D }
9110   }
9111 }

```

(End of definition for `\c_sys_timestamp_str`. This variable is documented on page 76.)

52.2.3 Random numbers

`\sys_rand_seed:` Unpack the primitive.

```

9112 \__sys_everyjob:n
9113 {
9114   \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D }
9115 }
```

(End of definition for `\sys_rand_seed:`. This function is documented on page 79.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

9116 \__sys_everyjob:n
9117 {
9118   \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9119     { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9120 }
```

(End of definition for `\sys_gset_rand_seed:n`. This function is documented on page 79.)

`\sys_timer:` In LuaTeX, create a pseudo-primitive, otherwise try to locate the real primitive. The elapsed time will be available if this succeeds.

```

\__sys_elapsedtime:
\sys_if_timer_exist_p:
\sys_if_timer_exist:TF
9121 </code>
9122 <lua>
9123   local gettimeofday = os.gettimeofday
9124   local epoch = gettimeofday() - os.clock()
9125   local write = tex.write
9126   local tointeger = math.tointeger
9127   luacmd('__sys_elapsedtime:', function()
9128     write(tointeger((gettimeofday() - epoch)*65536 // 1))
9129   end, 'global')
9130 </lua>
9131 <*code>
9132 \cs_new:Npe \sys_timer:
9133 {
9134   \sys_if_engine luatex:TF
9135     { \exp_not:N \__sys_elapsedtime: }
9136     { \exp_not:N \int_value:w \exp_not:N \tex_elapsedtime:D }
9137 }
```

(End of definition for `\sys_timer:`, `\__sys_elapsedtime:`, and `\sys_if_timer_exist:TF`. These functions are documented on page 78.)

52.2.4 Access to the shell

`\c_sys_shell_escape_int` Expose the engine's shell escape status to the user.

```

9138 \__sys_everyjob:n
9139 {
9140   \int_const:Nn \c_sys_shell_escape_int
9141     {
9142       \sys_if_engine luatex:TF
9143         {
9144           \tex_directlua:D
```

```

9145         { tex.sprint(status.shell_escape~or~os.execute()) }
9146     }
9147     { \tex_shellescape:D }
9148 }
9149 }

```

(End of definition for `\c_sys_shell_escape_int`. This variable is documented on page 80.)

`\sys_if_shell_p:` Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```

\sys_if_shell_unrestricted_p:
\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF
9150 \__sys_everyjob:n
9151 {
9152     \__sys_const:nn { sys_if_shell }
9153     { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9154     \__sys_const:nn { sys_if_shell_unrestricted }
9155     { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9156     \__sys_const:nn { sys_if_shell_restricted }
9157     { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9158 }

```

(End of definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 80.)

52.3 System queries

`\sys_get_query:nN` Calling the query system is quite straight-forward: most of the effort is in making the read-back catcode-safe. We also want to trim off the trailing $\sim$ from the last line.

```

\sys_get_query:nnN
\sys_get_query:nnnN
\sys_get_query:nnnnN
\__sys_get_query_auxi:nnnN
\__sys_get_query_auxi:neeN
\__sys_get_query_auxii:nnnN
\__sys_get_query_auxii:neeN
9159 \cs_new_protected:Npn \sys_get_query:nN #1#2
9160 { \sys_get_query:nnnN {#1} { } { } #2 }
9161 \cs_new_protected:Npn \sys_get_query:nnN #1#2#3
9162 { \sys_get_query:nnnN {#1} { } {#2} #3 }
9163 \cs_new_protected:Npn \sys_get_query:nnnN #1#2#3#4
9164 {
9165     \tl_clear:N #4
9166     \__sys_get_query_auxi:neeN {#1} {#2} {#3} #4
9167 }
9168 \cs_new:Npn \__sys_get_query_auxi:nnnN #1#2#3#4
9169 {
9170     \__sys_get_query_auxii:neeN {#1}
9171     { \tl_if_blank:nF {#2} { \tl_to_str:n { ~ #2 } } }
9172     {
9173         \tl_if_blank:nF {#3}
9174         {
9175             \c_space_tl
9176             \sys_if_shell_restricted:F '
9177             \tl_to_str:n {#3}
9178             \sys_if_shell_restricted:F '
9179         }
9180     }
9181     #4
9182 }
9183 \cs_generate_variant:Nn \__sys_get_query_auxi:nnnN { nee }

```

```

9184 \cs_new_protected:Npn \__sys_get_query_auxii:nnnN #1#2#3#4
9185 {
9186   \sys_if_shell:T
9187   {
9188     \sys_get_shell:nnN
9189     { l3sys-query~#1 #2 #3 }
9190     {
9191       \int_step_inline:nnn { 0 } { 'A - 1 }
9192       { \char_set_catcode_other:n {##1} }
9193       \int_step_inline:nnn { 'Z + 1 } { 'a - 1 }
9194       { \char_set_catcode_other:n {##1} }
9195       \int_step_inline:nnn { 'z + 1 } { 127 }
9196       { \char_set_catcode_other:n {##1} }
9197       \char_set_catcode_active:n { '\ }
9198       \tex_endlinechar:D 13 \scan_stop:
9199     }
9200     \l__sys_tmp_tl
9201     \tl_if_empty:NF \l__sys_tmp_tl
9202     {
9203       \exp_after:wN \__sys_get_query:Nw \exp_after:wN #4
9204       \l__sys_tmp_tl \q_stop
9205     }
9206   }
9207 }
9208 \cs_generate_variant:Nn \__sys_get_query_auxii:nnnN { nee }
9209 \group_begin:
9210   \tex_lccode:D '\* = 13 \scan_stop:
9211   \tex_lowercase:D
9212   {
9213     \group_end:
9214     \cs_new_protected:Npn \__sys_get_query:Nw #1#2 * \q_stop
9215   }
9216   { \tl_set:Nn #1 {#2} }

```

(End of definition for `\sys_get_query:nN` and others. These functions are documented on page 81.)

`\sys_split_query:nN`
`\sys_split_query:nnN`
`\sys_split_query:nnnN`

A wrapper for convenience.

```

9217 \cs_new_protected:Npn \sys_split_query:nN #1#2
9218 { \sys_split_query:nnnN {#1} { } { } #2 }
9219 \cs_new_protected:Npn \sys_split_query:nnN #1#2#3
9220 { \sys_split_query:nnnN {#1} { } {#2} #3 }
9221 \group_begin:
9222   \tex_lccode:D '\* = 13 \scan_stop:
9223   \tex_lowercase:D
9224   {
9225     \group_end:
9226     \cs_new_protected:Npn \sys_split_query:nnnN #1#2#3#4
9227     {
9228       \seq_clear:N #4
9229       \sys_get_query:nnnN {#1} {#2} {#3} \l__sys_tmp_tl
9230       \tl_if_empty:NF \l__sys_tmp_tl
9231       { \seq_set_split:NnV #4 * \l__sys_tmp_tl }
9232     }
9233   }

```

(End of definition for `\sys_split_query:nN`, `\sys_split_query:nnN`, and `\sys_split_query:nnnN`. These functions are documented on page 81.)

52.3.1 Held over from l3file

`\g_file_curr_name_str` See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get “tidied up”.

```
9234 \__sys_everyjob:n
9235   { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }
```

(End of definition for `\g_file_curr_name_str`. This variable is documented on page 102.)

52.4 Last-minute code

`\sys_finalize:` A simple hook to finalize the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```
\__sys_finalize:n
\g_sys_finalize_tl
9236 \cs_new_protected:Npn \sys_finalize:
9237   {
9238     \__kernel_sys_everyjob:
9239     \tl_use:N \g__sys_finalize_tl
9240     \tl_gclear:N \g__sys_finalize_tl
9241   }
9242 \cs_new_protected:Npn \__sys_finalize:n #1
9243   { \tl_gput_right:Nn \g__sys_finalize_tl {#1} }
9244 \tl_new:N \g__sys_finalize_tl
```

(End of definition for `\sys_finalize:`, `\__sys_finalize:n`, and `\g_sys_finalize_tl`. This function is documented on page 82.)

52.4.1 Detecting the output

`\sys_if_output_dvi_p:` This is a simple enough concept: the two views here are complementary.

```
\sys_if_output_dvi:TF
\sys_if_output_pdf_p:
\sys_if_output_pdf:TF
\c_sys_output_str
9245 \__sys_finalize:n
9246   {
9247     \str_const:Ne \c_sys_output_str
9248     {
9249       \int_compare:nNnTF
9250         { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9251         { pdf }
9252         { dvi }
9253     }
9254     \__sys_const:nn { sys_if_output_dvi }
9255     { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9256     \__sys_const:nn { sys_if_output_pdf }
9257     { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9258   }
```

(End of definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 78.)

52.4.2 Configurations

`\g__sys_backend_tl` As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9259 \tl_new:N \g__sys_backend_tl
9260 \__sys_finalize:n
9261 {
9262   \__kernel_tl_gset:Nx \g__sys_backend_tl
9263   {
9264     \sys_if_engine_xetex:TF
9265     { xetex }
9266     {
9267       \sys_if_output_pdf:TF
9268       {
9269         \sys_if_engine_pdftex:TF
9270         { pdftex }
9271         { luatex }
9272       }
9273       { dvips }
9274     }
9275   }
9276 }
```

If there is a class option set, and recognized, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9277 \__sys_finalize:n
9278 {
9279   \cs_if_exist:NT \@classoptionslist
9280   {
9281     \cs_if_eq:NNF \@classoptionslist \scan_stop:
9282     {
9283       \clist_map_inline:Nn \@classoptionslist
9284       {
9285         \str_case:nnT {#1}
9286         {
9287           { dvipdfmx }
9288           { \tl_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9289           { dvips }
9290           { \tl_gset:Nn \g__sys_backend_tl { dvips } }
9291           { dvisvgm }
9292           { \tl_gset:Nn \g__sys_backend_tl { dvisvgm } }
9293           { pdftex }
9294           { \tl_gset:Nn \g__sys_backend_tl { pdfmode } }
9295           { xetex }
9296           { \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9297         }
9298         { \clist_remove_all:Nn \@unusedoptionlist {#1} }
9299       }
9300     }
9301   }
9302 }
```

(End of definition for `\g__sys_backend_tl`.)

```

9303 \</code>
```


Chapter 53

l3msg implementation

```
9304 \*code\
9305 \@@=msg\

\l__msg_tmp_tl A general scratch for the module.
9306 \tl_new:N \l__msg_tmp_tl
(End of definition for \l__msg_tmp_tl.)

\l__msg_name_str Used to save module info when creating messages.
\l__msg_text_str
9307 \str_new:N \l__msg_name_str
9308 \str_new:N \l__msg_text_str
(End of definition for \l__msg_name_str and \l__msg_text_str.)
```

53.1 Internal auxiliaries

```
\s__msg_mark Internal scan marks.
\s__msg_stop
9309 \scan_new:N \s__msg_mark
9310 \scan_new:N \s__msg_stop
(End of definition for \s__msg_mark and \s__msg_stop.)

\_msg_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
9311 \cs_new:Npn \_msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
(End of definition for \_msg_use_none_delimit_by_s_stop:w.)
```

53.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```
\c__msg_text_prefix_tl Locations for the text of messages.
\c__msg_more_text_prefix_tl
9312 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }
9313 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra-text~>~ }
```

(End of definition for \c__msg_text_prefix_tl and \c__msg_more_text_prefix_tl.)

\msg_if_exist_p:nn Test whether the control sequence containing the message text exists or not.
\msg_if_exist:nnTF

```

9314 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
9315 {
9316     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
9317     { \prg_return_true: } { \prg_return_false: }
9318 }
```

(End of definition for \msg_if_exist:nnTF. This function is documented on page 84.)

__msg_chk_if_free:nn This auxiliary is similar to __kernel_chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```

9319 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
9320 {
9321     \msg_if_exist:nnT {#1} {#2}
9322     {
9323         \msg_error:nnnn { msg } { already-defined }
9324         {#1} {#2}
9325     }
9326 }
```

(End of definition for __msg_chk_if_free:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

\msg_new:nnee
\msg_new:nnxx
\msg_new:nnn
\msg_new:nne
\msg_new:nnx
\msg_set:nnnn
\msg_set:nnn

```

9327 \cs_new_protected:Npn \msg_new:nnnn #1#2#3#4
9328 {
9329     \__msg_chk_free:nn {#1} {#2}
9330     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
9331     ##1##2##3##4 {#3}
9332     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9333     ##1##2##3##4 {#4}
9334 }
9335 \cs_generate_variant:Nn \msg_new:nnnn { nnee , nnxx }
9336 \cs_new_protected:Npn \msg_new:nnn #1#2#3
9337 { \msg_new:nnnn {#1} {#2} {#3} { } }
9338 \cs_generate_variant:Nn \msg_new:nnn { nne , nnx }
9339 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
9340 {
9341     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
9342     ##1##2##3##4 {#3}
9343     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9344     ##1##2##3##4 {#4}
9345 }
9346 \cs_new_protected:Npn \msg_set:nnn #1#2#3
9347 { \msg_set:nnnn {#1} {#2} {#3} { } }
```

(End of definition for \msg_new:nnnn and others. These functions are documented on page 84.)

53.3 Messages: support functions and text

```

\c__msg_coding_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl

9348 \tl_const:Nn \c__msg_coding_error_text_tl
9349 {
9350   This~is~a~coding~error.
9351   \\ \\
9352 }
9353 \tl_const:Nn \c__msg_continue_text_tl
9354 { Type~<return>~to~continue }
9355 \tl_const:Nn \c__msg_critical_text_tl
9356 { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
9357 \tl_const:Nn \c__msg_fatal_text_tl
9358 { This~is~a~fatal~error:~LaTeX~will~abort. }
9359 \tl_const:Nn \c__msg_help_text_tl
9360 { For~immediate~help~type~H~<return> }
9361 \tl_const:Nn \c__msg_no_info_text_tl
9362 {
9363   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
9364   \c__msg_return_text_tl
9365 }
9366 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
9367 \tl_const:Nn \c__msg_return_text_tl
9368 {
9369   \\ \\
9370   Try~typing~<return>~to~proceed.
9371   \\
9372   If~that~doesn't~work,~type~X~<return>~to~quit.
9373 }
9374 \tl_const:Nn \c__msg_trouble_text_tl
9375 {
9376   \\ \\
9377   More~errors~will~almost~certainly~follow: \\
9378   the~LaTeX~run~should~be~aborted.
9379 }

```

(End of definition for \c__msg_coding_error_text_tl and others.)

\msg_line_number: For writing the line number nicely. **\msg_line_context:** was set up earlier, so this is not new.

```

9380 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
9381 \cs_gset:Npn \msg_line_context:
9382 {
9383   \c__msg_on_line_text_tl
9384   \c_space_tl
9385   \msg_line_number:
9386 }

```

(End of definition for \msg_line_number: and \msg_line_context:. These functions are documented on page 85.)

53.4 Showing messages: low level mechanism

```

\__msg_interrupt:Nnnn
\__msg_no_more_text:nnnn

```

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```

9387 \cs_new_protected:Npn \__msg_interrupt:NnnnN #1#2#3#4#5
9388   {
9389     \str_set:Ne \l__msg_text_str { #1 {#2} }
9390     \str_set:Ne \l__msg_name_str { \msg_module_name:n {#2} }
9391     \cs_if_eq:cNTF
9392       { \c__msg_more_text_prefix_tl #2 / #3 }
9393       \__msg_no_more_text:nnnn
9394     {
9395       \__msg_interrupt_wrap:nnn
9396       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
9397       { \c__msg_continue_text_tl }
9398       {
9399         \c__msg_no_info_text_tl
9400         \tl_if_empty:NF #5
9401         { \ \ \ #5 }
9402       }
9403     }
9404     {
9405       \__msg_interrupt_wrap:nnn
9406       { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
9407       { \c__msg_help_text_tl }
9408       {
9409         \use:c { \c__msg_more_text_prefix_tl #2 / #3 } #4
9410         \tl_if_empty:NF #5
9411         { \ \ \ #5 }
9412       }
9413     }
9414   }
9415 \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }

```

(End of definition for `\__msg_interrupt:Nnnn` and `\__msg_no_more_text:nnnn`.)

```

\__msg_interrupt_wrap:nnn
\__msg_interrupt_text:n
\__msg_interrupt_more_text:n

```

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using e-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We have to split the main text into two parts as only the “message” itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```

9416 \cs_new_protected:Npn \__msg_interrupt_wrap:nnn #1#2#3
9417   {
9418     \iow_wrap:nnnN { \ \ #3 } { } { } \__msg_interrupt_more_text:n
9419     \group_begin:
9420       \int_sub:Nn \l_iow_line_count_int { 2 }
9421       \iow_wrap:nenN { \l__msg_text_str : ~ #1 }

```

```

9422     {
9423       ( \l__msg_name_str )
9424       \prg_replicate:nn
9425       {
9426         \str_count:N \l__msg_text_str
9427         - \str_count:N \l__msg_name_str
9428         + 2
9429       }
9430       { ~ }
9431     }
9432     { } \__msg_interrupt_text:n
9433     \iow_wrap:nnnN { \l__msg_tmp_tl \\\ #2 } { } { }
9434     \__msg_interrupt:n
9435   }
9436   \cs_new_protected:Npn \__msg_interrupt_text:n #1
9437   {
9438     \group_end:
9439     \tl_set:Nn \l__msg_tmp_tl {#1}
9440   }
9441   \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
9442   { \exp_args:Ne \tex_errhelp:D { #1 \iow_newline: } }

```

(End of definition for __msg_interrupt_wrap:nnn, __msg_interrupt_text:n, and __msg_interrupt_more_text:n.)

`\__msg_interrupt:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: $\text{\TeX}$ ’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {<spaces>}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `\__kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an *<integer variable>*, an integer *<value>*, and some *<code>*. It runs the *<code>* after ensuring that the *<integer variable>* takes the given *<value>*, then restores the former value of the *<integer variable>* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

9443 \group_begin:
9444   \char_set_lccode:nn { 38 } { 32 } % &
9445   \char_set_lccode:nn { 46 } { 32 } % .
9446   \char_set_lccode:nn { 123 } { 32 } % {
9447   \char_set_lccode:nn { 125 } { 32 } % }
9448   \char_set_catcode_active:N \&
9449   \tex_lowercase:D
9450   {
9451     \group_end:
9452     \cs_new_protected:Npn \__msg_interrupt:n #1
9453     {

```

```

9454 \iow_term:n { }
9455 \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
9456 {
9457   \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9458   {
9459     \group_begin:
9460     \cs_set_protected:Npn &
9461     {
9462       \tex_errmessage:D
9463       {
9464         #1
9465         \use_none:n
9466         { ..... }
9467       }
9468     }
9469     \exp_after:wN
9470     \group_end:
9471     &
9472   }
9473 }
9474 }
9475 }

```

(End of definition for `\__msg_interrupt:n`.)

53.5 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled.

```

9476 \int_gset:Nn \tex_errorcontextlines:D { -1 }

```

`\msg_fatal_text:n` A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```

\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
\__msg_text:nn
\__msg_text:n
9477 \cs_new:Npn \msg_fatal_text:n #1
9478 {
9479   Fatal ~
9480   \msg_error_text:n {#1}
9481 }
9482 \cs_new:Npn \msg_critical_text:n #1
9483 {
9484   Critical ~
9485   \msg_error_text:n {#1}
9486 }
9487 \cs_new:Npn \msg_error_text:n #1
9488 { \__msg_text:nn {#1} { Error } }
9489 \cs_new:Npn \msg_warning_text:n #1
9490 { \__msg_text:nn {#1} { Warning } }
9491 \cs_new:Npn \msg_info_text:n #1
9492 { \__msg_text:nn {#1} { Info } }
9493 \cs_new:Npn \__msg_text:nn #1#2
9494 {
9495   \exp_args:Nf \__msg_text:n { \msg_module_type:n {#1} }
9496   \exp_args:Nf \__msg_text:n { \msg_module_name:n {#1} }

```

```

9497     #2
9498   }
9499   \cs_new:Npn \_msg_text:n #1
9500   {
9501     \tl_if_blank:nF {#1}
9502     { #1 ~ }
9503   }

```

(End of definition for `\msg_fatal_text:n` and others. These functions are documented on page 85.)

`\g_msg_module_name_prop` For storing public module information: the kernel data is set up in advance.
`\g_msg_module_type_prop`

```

9504   \prop_new:N \g_msg_module_name_prop
9505   \prop_new:N \g_msg_module_type_prop
9506   \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End of definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 84.)

`\msg_module_type:n` Contextual footer information, with the potential to give modules an alternative name.

```

9507   \cs_new:Npn \msg_module_type:n #1
9508   {
9509     \prop_if_in:NnTF \g_msg_module_type_prop {#1}
9510     { \prop_item:Nn \g_msg_module_type_prop {#1} }
9511     { Package }
9512   }

```

(End of definition for `\msg_module_type:n`. This function is documented on page 84.)

`\msg_module_name:n` Contextual footer information, with the potential to give modules an alternative name.
`\msg_see_documentation_text:n`

```

9513   \cs_new:Npn \msg_module_name:n #1
9514   {
9515     \prop_if_in:NnTF \g_msg_module_name_prop {#1}
9516     { \prop_item:Nn \g_msg_module_name_prop {#1} }
9517     {#1}
9518   }
9519   \cs_new:Npn \msg_see_documentation_text:n #1
9520   {
9521     See-the~ \msg_module_name:n {#1} ~
9522     documentation-for-further-information.
9523   }

```

(End of definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 84.)

`\_msg_class_new:nn`

```

9524   \group_begin:
9525   \cs_set_protected:Npn \_msg_class_new:nn #1#2
9526   {
9527     \prop_new:c { l_msg_redirect_ #1 _prop }
9528     \cs_new_protected:cpn { _msg_ #1 _code:nnnnnn }
9529     ##1##2##3##4##5##6 {#2}
9530     \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9531     {
9532       \use:e
9533       {

```

```

9534         \exp_not:n { \_msg_use:nnnnnnn {#1} {##1} {##2} }
9535         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9536         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9537     }
9538 }
9539 \cs_new_protected:cpe { msg_ #1 :nnnnn } ##1##2##3##4##5
9540 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9541 \cs_new_protected:cpe { msg_ #1 :nnnn } ##1##2##3##4
9542 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9543 \cs_new_protected:cpe { msg_ #1 :nnn } ##1##2##3
9544 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9545 \cs_new_protected:cpe { msg_ #1 :nn } ##1##2
9546 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9547 \cs_generate_variant:cn { msg_ #1 :nnn }
9548 { nnV , nne , nnx }
9549 \cs_generate_variant:cn { msg_ #1 :nnnn }
9550 { nnVV , nnVn , nnnV , nnne , nnnx , nnee , nxxx }
9551 \cs_generate_variant:cn { msg_ #1 :nnnnn }
9552 { nnnee , nnnxx , nneee , nxxxx }
9553 \cs_generate_variant:cn { msg_ #1 :nnnnnn } { nneeee , nxxxxx }
9554 }

```

(End of definition for _msg_class_new:nn.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message TeX bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```

\msg_fatal:nnnnnn 9555 \_msg_class_new:nn { fatal }
\msg_fatal:nneeee 9556 {
\msg_fatal:nnxxxx 9557     \_msg_interrupt:NnnnN
\msg_fatal:nnnee 9558     \msg_fatal_text:n {#1} {#2}
\msg_fatal:nnnxx 9559     { {#3} {#4} {#5} {#6} }
\msg_fatal:nnnn 9560     \c_msg_fatal_text_tl
\msg_fatal:nnVV 9561     \_msg_fatal_exit:
\msg_fatal:nnVn 9562 }
\msg_fatal:nnnV 9563 \cs_new_protected:Npn \_msg_fatal_exit:
\msg_fatal:nnee 9564 {
\msg_fatal:nnxx 9565     \tex_batchmode:D
\msg_fatal:nnnx 9566     \tex_read:D -1 to \l_msg_tmp_tl
\msg_fatal:nnne 9567 }

```

(End of definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 87.)

```

\msg_fatal:nnV 9568 \_msg_class_new:nn { critical }
\msg_fatal:nnnnnn 9569 {
\msg_fatal:nnee 9570     \_msg_interrupt:NnnnN
\msg_fatal:nneeee 9571     \msg_critical_text:n {#1} {#2}
\msg_fatal:nnxx 9572     { {#3} {#4} {#5} {#6} }
\msg_fatal:nnnnnn 9573     \c_msg_critical_text_tl
\msg_fatal:nnnee 9574     \tex_endinput:D
\msg_fatal:nnnxx 9575 }

```

Not quite so bad: just end the current file.

(End of definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 87.)


```

\msg_error:nnnnnn
\msg_error:nneeee
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nneee
\msg_error:nnxxx
\msg_error:nnnee
\msg_error:nnnxx
\msg_error:nnnn
\msg_error:nnVV
\msg_error:nnVn
\msg_error:nnnV
\msg_error:nnee
\msg_error:nnxx
\msg_error:nnnx
\__msg_info_aux:Nnnnnnn
\msg_error:nnee
\msg_warning:nnnnn
\msg_warning:nneeee
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_error:nnx
\msg_warning:nnee
\msg_warning:nnnxx
\msg_warning:nnnn
\msg_warning:nnVV
\msg_warning:nnVn
\msg_warning:nnnV
\msg_warning:nnee
\msg_warning:nnxx
\msg_warning:nnnx
\msg_warning:nnne
\msg_warning:nnn
\msg_warning:nnV
\msg_warning:nne
\msg_warning:nnx
\msg_warning:nn
\msg_note:nnnnnn
\msg_note:nneeee
\msg_note:nnxxxx
\msg_note:nnnnn
\msg_note:nneee
\msg_note:nnxxx
\msg_note:nnnee
\msg_note:nnnxx
\msg_note:nnnn
\msg_note:nnVV
\msg_note:nnVn
\msg_note:nnnV
\msg_note:nnee
\msg_note:nnxx
\msg_note:nnnx
\msg_note:nnne
\msg_note:nnn
\msg_note:nnV
\msg_note:nne
\msg_note:nnx
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee

```

For an error, the interrupt routine is called. We check if there is a “more text” by comparing that control sequence with a permanently empty text. We have to undefine the bootstrap versions here.

```

9576 \cs_undefine:N \msg_error:nnee
9577 \cs_undefine:N \msg_error:nnee
9578 \cs_undefine:N \msg_error:nn
9579 \__msg_class_new:nn { error }
9580 {
9581   \__msg_interrupt:NnnnN
9582   \msg_error_text:n {#1} {#2}
9583   { {#3} {#4} {#5} {#6} }
9584   \c_empty_tl
9585 }

```

(End of definition for \msg_error:nnnnnn and others. These functions are documented on page 87.)

Warnings and information messages have no decoration. Warnings are printed to the terminal while information can either go to the log or both log and terminal.

```

9586 \cs_new_protected:Npn \__msg_info_aux:Nnnnnnnn #1#2#3#4#5#6#7#8
9587 {
9588   \str_set:Ne \l__msg_text_str { #2 {#3} }
9589   \str_set:Ne \l__msg_name_str { \msg_module_name:n {#3} }
9590   #1 { }
9591   \iow_wrap:nenN
9592   {
9593     \l__msg_text_str : ~
9594     \use:c { \c__msg_text_prefix_tl #3 / #4 } {#5} {#6} {#7} {#8}
9595   }
9596   {
9597     ( \l__msg_name_str )
9598     \prg_replicate:nn
9599     {
9600       \str_count:N \l__msg_text_str
9601       - \str_count:N \l__msg_name_str
9602     }
9603     { ~ }
9604   }
9605   { } #1
9606   #1 { }
9607 }
9608 \__msg_class_new:nn { warning }
9609 {
9610   \__msg_info_aux:Nnnnnnnn \iow_term:n \msg_warning_text:n
9611   {#1} {#2} {#3} {#4} {#5} {#6}
9612 }
9613 \__msg_class_new:nn { note }
9614 {
9615   \__msg_info_aux:Nnnnnnnn \iow_term:n \msg_info_text:n
9616   {#1} {#2} {#3} {#4} {#5} {#6}
9617 }
9618 \__msg_class_new:nn { info }
9619 {
9620   \__msg_info_aux:Nnnnnnnn \iow_log:n \msg_info_text:n
9621   {#1} {#2} {#3} {#4} {#5} {#6}

```

641

```

9655 { \_msg_show:nn {#1} {#2} }
9656 \cs_new_protected:Npn \_msg_show:nn #1#2
9657 {
9658   \tl_if_empty:nF {#1}
9659   { \exp_args:No \iow_term:n { \use_none:n #1 } }
9660   \tl_set:Nn \l__msg_tmp_tl {#2}
9661   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
9662   {
9663     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9664     {
9665       \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9666       { \exp_after:wN \l__msg_tmp_tl }
9667     }
9668   }
9669 }

```

(End of definition for `\msg_show:nnnnnn` and others. These functions are documented on page 90.)

End the group to eliminate `\_msg_class_new:nn`.

```

9670 \group_end:

```

```

\_msg_show_item:n
\_msg_show_item_unbraced:n
\_msg_show_item:nn
\_msg_show_item_unbraced:nn

```

Each item in the variable is formatted using one of the following functions. We cannot use `\` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages. We need to use `^^J` here directly as `l3file` is not yet loaded.

```

9671 \cs_new:Npe \msg_show_item:n #1
9672 { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
9673 \cs_new:Npe \msg_show_item_unbraced:n #1
9674 { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
9675 \cs_new:Npe \msg_show_item:nn #1#2
9676 {
9677   ^^J > \use:nn { ~ } { ~ }
9678   \exp_not:N \tl_to_str:n { {#1} }
9679   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
9680   \exp_not:N \tl_to_str:n { {#2} }
9681 }
9682 \cs_new:Npe \msg_show_item_unbraced:nn #1#2
9683 {
9684   ^^J > \use:nn { ~ } { ~ }
9685   \exp_not:N \tl_to_str:n {#1}
9686   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
9687   \exp_not:N \tl_to_str:n {#2}
9688 }

```

(End of definition for `\msg_show_item:n` and others. These functions are documented on page 90.)

```

\_msg_class_chk_exist:nT

```

Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

9689 \cs_new:Npn \_msg_class_chk_exist:nT #1
9690 {
9691   \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9692   { \msg_error:nnn { msg } { class-unknown } {#1} }
9693 }

```

(End of definition for _msg_class_chk_exist:nT.)

_msg_class_tl Support variables needed for the redirection system.

```

\_msg\_current\_class\_tl 9694 \tl\_new:N \\_msg\_class\_tl
                        9695 \tl\_new:N \\_msg\_current\_class\_tl

```

(End of definition for _msg_class_tl and _msg_current_class_tl.)

_msg_redirect_prop For redirection of individually-named messages

```
9696 \prop\_new:N \\_msg\_redirect\_prop
```

(End of definition for _msg_redirect_prop.)

_msg_hierarchy_seq During redirection, split the message name into a sequence: {/module/submodule}, {/module}, and {}.

```
9697 \seq\_new:N \\_msg\_hierarchy\_seq
```

(End of definition for _msg_hierarchy_seq.)

_msg_class_loop_seq Classes encountered when following redirections to check for loops.

```
9698 \seq\_new:N \\_msg\_class\_loop\_seq
```

(End of definition for _msg_class_loop_seq.)

_msg_use:nnnnnnn

_msg_use_redirect_name:n

_msg_use_hierarchy:nwWN

_msg_use_redirect_module:n

_msg_use_code:

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to _msg_use_code: is similar to \tl_set:Nn. The message is eventually produced with whatever _msg_class_tl is when _msg_use_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```

9699 \cs\_new\_protected:Npn \_msg\_use:nnnnnnn #1#2#3#4#5#6#7
9700 {
9701   \cs\_if\_exist\_use:N \conditionally@traceoff
9702   \msg\_if\_exist:nnTF {#2} {#3}
9703   {
9704     \_msg\_class\_chk\_exist:nT {#1}
9705     {
9706       \tl\_set:Nn \_msg\_current\_class\_tl {#1}
9707       \cs\_set\_protected:Npe \_msg\_use\_code:
9708       {
9709         \exp\_not:n
9710         {
9711           \use:c { \_msg\_ \_msg\_class\_tl \_code:nnnnnn }
9712           {#2} {#3} {#4} {#5} {#6} {#7}
9713         }
9714       }
9715       \_msg\_use\_redirect\_name:n { #2 / #3 }
9716     }
9717   }
9718   { \msg\_error:nnnn { msg } { unknown } {#2} {#3} }
9719   \cs\_if\_exist\_use:N \conditionally@traceon
9720 }
9721 \cs\_new\_protected:Npn \_msg\_use\_code: { }

```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into $\langle module \rangle$, $\langle submodule \rangle$ and $\langle message \rangle$ (with an arbitrary number of slashes), and store $\{/module/submodule\}$, $\{/module\}$ and $\{\}$ into $\backslash l\_msg\_hierarchy\_seq$. We then map through this sequence, applying the most specific redirection.

```

9722 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
9723 {
9724   \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9725   { \__msg_use_code: }
9726   {
9727     \seq_clear:N \l__msg_hierarchy_seq
9728     \__msg_use_hierarchy:nwwN { }
9729     #1 \s__msg_mark \__msg_use_hierarchy:nwwN
9730     / \s__msg_mark \__msg_use_none_delimit_by_s_stop:w
9731     \s__msg_stop
9732     \__msg_use_redirect_module:n { }
9733   }
9734 }
9735 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
9736 {
9737   \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9738   #4 { #1 / #2 } #3 \s__msg_mark #4
9739 }

```

At this point, the items of $\backslash l\_msg\_hierarchy\_seq$ are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of $\backslash \_msg\_use\_redirect\_module:n$ are not attempted. This argument is empty for a class redirection, $/module$ for a module redirection, etc. Loop through the sequence to find the most specific redirection, with module $\##1$. The loop is interrupted after testing for a redirection for $\##1$ equal to the argument $\#1$ (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as $\##1$.

```

9740 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9741 {
9742   \seq_map_inline:Nn \l__msg_hierarchy_seq
9743   {
9744     \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9745     {##1} \l__msg_class_tl
9746     {
9747       \seq_map_break:n
9748       {
9749         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9750         { \__msg_use_code: }
9751         {
9752           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9753           \__msg_use_redirect_module:n {##1}
9754         }
9755       }
9756     }
9757   }
9758   \str_if_eq:nnT {##1} {#1}

```

```

9759         {
9760             \tl_set_eq:Nn \l__msg_class_tl \l__msg_current_class_tl
9761             \seq_map_break:n { \__msg_use_code: }
9762         }
9763     }
9764 }
9765 }

```

(End of definition for `\__msg_use:nnnnnnn` and others.)

`\msg_redirect_name:nnn` Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

9766 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9767 {
9768     \tl_if_empty:nTF {#3}
9769     { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9770     {
9771         \__msg_class_chk_exist:nT {#3}
9772         { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9773     }
9774 }

```

(End of definition for `\msg_redirect_name:nnn`. This function is documented on page 92.)

`\msg_redirect_class:nn` If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

`\msg_redirect_module:nnn`
`\__msg_redirect:nnn`
`\__msg_redirect_loop_chk:nnn`
`\__msg_redirect_loop_list:n`

```

9775 \cs_new_protected:Npn \msg_redirect_class:nn
9776 { \__msg_redirect:nnn { } }
9777 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9778 { \__msg_redirect:nnn { / #1 } }
9779 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9780 {
9781     \__msg_class_chk_exist:nT {#2}
9782     {
9783         \tl_if_empty:nTF {#3}
9784         { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9785         {
9786             \__msg_class_chk_exist:nT {#3}
9787             {
9788                 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9789                 \tl_set:Nn \l__msg_current_class_tl {#2}
9790                 \seq_clear:N \l__msg_class_loop_seq
9791                 \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9792             }
9793         }
9794     }
9795 }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to

itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

9796 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9797 {
9798   \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9799   \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
9800   {
9801     \str_if_eq:VnF \l__msg_class_tl {#1}
9802     {
9803       \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9804       {
9805         \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9806         \msg_warning:nneeee
9807         { msg } { redirect-loop }
9808         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9809         { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
9810         {#3}
9811         {
9812           \seq_map_function:NN \l__msg_class_loop_seq
9813             \__msg_redirect_loop_list:n
9814             { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9815         }
9816       }
9817       { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9818     }
9819   }
9820 }
9821 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9822 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End of definition for `\msg_redirect_class:nn` and others. These functions are documented on page 92.)

53.6 Kernel-specific functions

`\__kernel_msg_show_eval:Nn` A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

9823 \cs_new_protected:Npn \__kernel_msg_show_eval:Nn #1#2
9824 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
9825 \cs_new_protected:Npn \__kernel_msg_log_eval:Nn #1#2
9826 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
9827 \cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }

```

(End of definition for `\_kernel_msg_show_eval:Nn`, `\_kernel_msg_log_eval:Nn`, and `\_msg_show_eval:nnN`.)

These are all retained purely for older xparse support.

```
\_kernel_msg_new:nnnn
\_kernel_msg_new:nnn
9828 \cs_new_protected:Npn \_kernel_msg_new:nnnn #1
9829   { \msg_new:nnnn { LaTeX / #1 } }
9830 \cs_new_protected:Npn \_kernel_msg_new:nnn #1
9831   { \msg_new:nnn { LaTeX / #1 } }
```

(End of definition for `\_kernel_msg_new:nnnn` and `\_kernel_msg_new:nnn`.)

```
\_kernel_msg_info:nnxx
\_kernel_msg_warning:nnx
\_kernel_msg_warning:nnxx
\_kernel_msg_error:nnx
\_kernel_msg_error:nnxx
\_kernel_msg_error:nnxxx
9832 \cs_new_protected:Npn \_kernel_msg_info:nnxx #1
9833   { \msg_info:nnee { LaTeX / #1 } }
9834 \cs_new_protected:Npn \_kernel_msg_warning:nnx #1
9835   { \msg_warning:nne { LaTeX / #1 } }
9836 \cs_new_protected:Npn \_kernel_msg_warning:nnxx #1
9837   { \msg_warning:nnee { LaTeX / #1 } }
9838 \cs_new_protected:Npn \_kernel_msg_error:nnx #1
9839   { \msg_error:nne { LaTeX / #1 } }
9840 \cs_new_protected:Npn \_kernel_msg_error:nnxx #1
9841   { \msg_error:nnee { LaTeX / #1 } }
9842 \cs_new_protected:Npn \_kernel_msg_error:nnxxx #1
9843   { \msg_error:nnee { LaTeX / #1 } }
```

(End of definition for `\_kernel_msg_info:nnxx` and others.)

```
\_kernel_msg_expandable_error:nnn
\_kernel_msg_expandable_error:nnf
\_kernel_msg_expandable_error:nnff
9844 \cs_new:Npn \_kernel_msg_expandable_error:nnn #1
9845   { \msg_expandable_error:nnn { LaTeX / #1 } }
9846 \cs_new:Npn \_kernel_msg_expandable_error:nnf #1
9847   { \msg_expandable_error:nnf { LaTeX / #1 } }
9848 \cs_new:Npn \_kernel_msg_expandable_error:nnff #1
9849   { \msg_expandable_error:nnff { LaTeX / #1 } }
```

(End of definition for `\_kernel_msg_expandable_error:nnn` and `\_kernel_msg_expandable_error:nnff`.)

53.7 Internal messages

Error messages needed to actually implement the message system itself.

```
9850 \msg_new:nnnn { msg } { already-defined }
9851   { Message~'#2'~for~module~'#1'~already-defined. }
9852   {
9853     \c_msg_coding_error_text_tl
9854     LaTeX~was~asked~to~define~a~new~message~called~'#2'\
9855     by~the~module~'#1':~this~message~already~exists.
9856     \c_msg_return_text_tl
9857   }
9858 \msg_new:nnnn { msg } { unknown }
9859   { Unknown~message~'#2'~for~module~'#1'. }
9860   {
9861     \c_msg_coding_error_text_tl
```



```

9862 LaTeX~was~asked~to~display~a~message~called~'#2'\
9863 by~the~module~'#1':~this~message~does~not~exist.
9864 \c__msg_return_text_tl
9865 }
9866 \msg_new:nnnn { msg } { class-unknown }
9867 { Unknown~message~class~'#1'. }
9868 {
9869 LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
9870 this~was~never~defined.
9871 \c__msg_return_text_tl
9872 }
9873 \msg_new:nnnn { msg } { redirect-loop }
9874 {
9875 Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
9876 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9877 }
9878 {
9879 Adding~the~message~redirection~ {#1} ~=>~ {#2}
9880 \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
9881 created~an~infinite~loop\\\
9882 \iow_indent:n { #4 \\\ }
9883 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

9884 \msg_new:nnnn { kernel } { bad-number-of-arguments }
9885 { Function~'#1'~cannot~be~defined~with~#2~arguments. }
9886 {
9887 \c__msg_coding_error_text_tl
9888 LaTeX~has~been~asked~to~define~a~function~'#1'~with~
9889 #2~arguments.~
9890 TeX~allows~between~0~and~9~arguments~for~a~single~function.
9891 }
9892 \msg_new:nnnn { kernel } { command-already-defined }
9893 { Control~sequence~#1~already~defined. }
9894 {
9895 \c__msg_coding_error_text_tl
9896 LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
9897 but~this~name~has~already~been~used~elsewhere. \ \
9898 The~current~meaning~is:\
9899 \ \ #2
9900 }
9901 \msg_new:nnnn { kernel } { command-not-defined }
9902 { Control~sequence~#1~undefined. }
9903 {
9904 \c__msg_coding_error_text_tl
9905 LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\
9906 this~has~not~been~defined~yet.
9907 }
9908 \msg_new:nnnn { kernel } { empty-search-pattern }
9909 { Empty~search~pattern. }
9910 {
9911 \c__msg_coding_error_text_tl
9912 LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
9913 would~lead~to~an~infinite~loop!
9914 }

```

```

9915 \msg_new:nnnn { kernel } { non-base-function }
9916 { Function~'#1'~is-not-a-base-function }
9917 {
9918   \c_msg_coding_error_text_tl
9919   Functions~defined-through~\iow_char:N\cs_new:Nn~must~have~
9920   a~signature~consisting-of~only~normal~arguments~'N'~and~'n'.~
9921   The~signature~'#2'~of~'#1'~contains~other~arguments~'#3'.~
9922   To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
9923   and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
9924 }
9925 \msg_new:nnnn { kernel } { missing-colon }
9926 { Function~'#1'~contains-no~':'. }
9927 {
9928   \c_msg_coding_error_text_tl
9929   Code-level~functions~must~contain~': '~to~separate~the~
9930   argument~specification~from~the~function~name.~This~is~
9931   needed~when~defining~conditionals~or~variants,~or~when~building~a~
9932   parameter~text~from~the~number~of~arguments~of~the~function.
9933 }
9934 \msg_new:nnnn { kernel } { overflow }
9935 { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
9936 {
9937   An~attempt~was~made~to~store~'#3~
9938   \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
9939   The~largest~allowed~value~#4~will~be~used~instead.
9940 }
9941 \msg_new:nnnn { kernel } { out-of-bounds }
9942 { Access~to~an~entry~beyond~an~array's~bounds. }
9943 {
9944   An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
9945   array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
9946 }
9947 \msg_new:nnnn { kernel } { protected-predicate }
9948 { Predicate~'~'#1'~must~be~expandable. }
9949 {
9950   \c_msg_coding_error_text_tl
9951   LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
9952   Only~expandable~tests~can~have~a~predicate~version.
9953 }
9954 \msg_new:nnn { kernel } { randint-backward-range }
9955 { Wrong~order~of~bounds~in~\iow_char:N\int_rand:nn{#1}{#2}. }
9956 \msg_new:nnnn { kernel } { conditional-base-undefined }
9957 { Undefined~conditional~base~function~'#1'. }
9958 {
9959   \c_msg_coding_error_text_tl
9960   LaTeX~has~been~asked~to~define~a~variant~of~the~conditional~'#1',~
9961   but~the~latter~is~not~defined.
9962 }
9963 \msg_new:nnnn { kernel } { conditional-form-unknown }
9964 { Conditional~form~'#1'~for~function~'#2'~unknown. }
9965 {
9966   \c_msg_coding_error_text_tl
9967   LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
9968   the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.

```

```

9969 }
9970 \msg_new:nnnn { kernel } { variant-too-long }
9971 { Variant-form~'#1'~longer-than-base-signature-of~'#2'. }
9972 {
9973   \c__msg_coding_error_text_tl
9974   LaTeX-has-been-asked-to-create-a-variant-of-the-function~'#2'~
9975   with-a-signature-starting-with~'#1',~but-that-is-longer-than~
9976   the-signature-(part-after-the-colon)-of~'#2'.
9977 }
9978 \msg_new:nnnn { kernel } { invalid-variant }
9979 { Variant-form~'#1'~invalid-for-base-form~'#2'. }
9980 {
9981   \c__msg_coding_error_text_tl
9982   LaTeX-has-been-asked-to-create-a-variant-of-the-function~'#2'~
9983   with-a-signature-starting-with~'#1',~but~cannot~change~an~argument~
9984   from~type~'#3'~to~type~'#4'.
9985 }
9986 \msg_new:nnnn { kernel } { invalid-exp-args }
9987 { Invalid-variant-specifier~'#1'~in~'#2'. }
9988 {
9989   \c__msg_coding_error_text_tl
9990   LaTeX-has-been-asked-to-create-an~\iow_char:N\\exp_args:N...~
9991   function-with-signature~'N#2'~but~'#1'~is-not-a-valid-argument~
9992   specifier.
9993 }
9994 \msg_new:nnn { kernel } { deprecated-variant }
9995 {
9996   Variant-form~'#1'~deprecated-for-base-form~'#2'.~
9997   One-should-not-change-an-argument-from-type~'#3'~to~type~'#4'
9998   \str_case:nnF {#3}
9999   {
10000     { n } { :~use-a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
10001     { N } { :~base-form-only~accepts-a-single-token-argument. }
10002     {#4} { :~base-form-is-already-a-variant. }
10003   } { . }
10004 }
10005 \msg_new:nnn { char } { active }
10006 { Cannot-generate-active-chars. }
10007 \msg_new:nnn { char } { invalid-catcode }
10008 { Invalid-catcode-for-char-generation. }
10009 \msg_new:nnn { char } { null-space }
10010 { Cannot-generate-null-char-as-a-space. }
10011 \msg_new:nnn { char } { out-of-range }
10012 { Charcode-requested-out-of-engine-range. }
10013 \msg_new:nnn { dim } { zero-unit }
10014 { Zero-unit-in-conversion. }
10015 \msg_new:nnnn { kernel } { quote-in-shell }
10016 { Quotes-in-shell-command~'#1'. }
10017 { Shell-commands-cannot-contain-quotes~("). }
10018 \msg_new:nnnn { keys } { no-property }
10019 { No-property-given-in-definition-of-key~'#1'. }
10020 {
10021   \c__msg_coding_error_text_tl
10022   Inside~\keys_define:nn each-key-name~

```

```

10023     needs-a~property:  \ \ \
10024     \iow_indent:n { #1 .<property> } \ \ \
10025     LaTeX~did~not~find~a~'. ' ~to~indicate~the~start~of~a~property.
10026   }
10027   \msg_new:nnnn { keys } { property-boolean-values-only }
10028   { The~property~'#1'~accepts~boolean~values~only. }
10029   {
10030     \c__msg_coding_error_text_tl
10031     The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10032   }
10033   \msg_new:nnnn { keys } { property-requires-value }
10034   { The~property~'#1'~requires~a~value. }
10035   {
10036     \c__msg_coding_error_text_tl
10037     LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'. \ \
10038     No~value~was~given~for~the~property,~and~one~is~required.
10039   }
10040   \msg_new:nnnn { keys } { property-unknown }
10041   { The~key~property~'#1'~is~unknown. }
10042   {
10043     \c__msg_coding_error_text_tl
10044     LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10045     this~property~is~not~defined.
10046   }
10047   \msg_new:nnnn { quark } { invalid-function }
10048   { Quark~test~function~'#1'~is~invalid. }
10049   {
10050     \c__msg_coding_error_text_tl
10051     LaTeX~has~been~asked~to~create~quark~test~function~'#1'~
10052     \tl_if_empty:nTF {#2}
10053       { but~that~name~ }
10054       { with~signature~'#2',~but~that~signature~ }
10055     is~not~valid.
10056   }
10057   \__kernel_msg_new:nnn { quark } { invalid }
10058   { Invalid~quark~variable~'#1'. }
10059   \msg_new:nnnn { scanmark } { already-defined }
10060   { Scan~mark~#1~already~defined. }
10061   {
10062     \c__msg_coding_error_text_tl
10063     LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
10064     but~this~name~has~already~been~used~for~a~scan~mark.
10065   }
10066   \msg_new:nnnn { seq } { item-too-large }
10067   { Sequence~'#1'~does~not~have~an~item~#3 }
10068   {
10069     An~attempt~was~made~to~push~or~pop~the~item~at~position~#3~
10070     of~'#1',~but~this~
10071     \int_compare:nTF { #3 = 0 }
10072       { position~does~not~exist. }
10073       { sequence~only~has~#2~item \int_compare:nF { #2 = 1 } {s}. }
10074   }
10075   \msg_new:nnnn { seq } { shuffle-too-large }
10076   { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }

```

```

10077 {
10078   TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
10079   toks~registers:~this~only~allows~to~shuffle~up~to~
10080   \int_use:N \c_max_register_int \ items.~
10081   The~list~will~not~be~shuffled.
10082 }
10083 \msg_new:nnnn { kernel } { variable-not-defined }
10084 { Variable~'#1~undefined. }
10085 {
10086   \c__msg_coding_error_text_tl
10087   LaTeX~has~been~asked~to~show~a~variable~'#1,~but~this~has~not~
10088   been~defined~yet.
10089 }
10090 \msg_new:nnnn { kernel } { bad-type }
10091 { Variable~'#1'~is~not~a~valid~#3. }
10092 {
10093   \c__msg_coding_error_text_tl
10094   The~variable~'#1'~with~\tl_if_empty:nTF {#4} {meaning} {value}\\\\
10095   \iow_indent:n {#2}\\\\
10096   should~be~a~#3~variable,~but~
10097   \tl_if_empty:nTF {#4}
10098     { it~is~not \str_if_eq:nnF {#3} { bool } { ~a~short~macro } . }
10099     {
10100       it~does~not~have~the~correct~
10101       \str_if_eq:nnTF {#2} {#4}
10102       { category~codes. }
10103       { internal~structure:\\\\\iow_indent:n {#4} }
10104     }
10105 }
10106 \msg_new:nnnn { prop } { bad-link }
10107 { Variable~'#1'~is~not~a~valid~(linked)~prop. }
10108 {
10109   \c__msg_coding_error_text_tl
10110   The~variable~'#1'~has~an~incorrect~internal~structure.~
10111   Its~internal~entry~'#2'~points~to~'#3',~whose~name~is~not~of~the~
10112   form~'#4~<key>'.
10113 }
10114 \msg_new:nnnn { clist } { non-clist }
10115 { Variable~'#1'~is~not~a~valid~clist. }
10116 {
10117   \c__msg_coding_error_text_tl
10118   The~variable~'#1'~with~value\\\\
10119   \iow_indent:n {#2}\\\\
10120   should~be~a~clist~variable,~but~it~includes~empty~or~blank~items~
10121   without~braces.
10122 }
10123 \msg_new:nnnn { prop } { misused }
10124 { A~property~list~was~misused. }
10125 {
10126   \c__msg_coding_error_text_tl
10127   A~property~list~variable~was~used~without~an~accessor~function.~
10128   It~
10129   \tl_if_empty:nTF {#1}
10130     { is~empty. }

```

```

10131     { contains~the~key~value~pairs \use_none:n #1 . }
10132   }
10133   \msg_new:nnnn { prop } { inner-make }
10134   { '#1'~ cannot~ be~ used~ in~ a~ group. }
10135   {
10136     \c__msg_coding_error_text_tl
10137     The~ command~ '#1'~ was~ applied~ to~ the~ property~ list~
10138     variable~ '#2', but~ the~ storage~ type~ can~ only~ be~ changed~
10139     at~ the~ outermost~ group~ level.
10140   }

```

Some errors only appear in expandable settings, hence don't need a “more-text” argument.

```

10141 \msg_new:nnn { kernel } { bad-exp-end-f }
10142   { Misused~\exp_end_continue_f:w or~:nw }
10143 \msg_new:nnn { kernel } { bad-variable }
10144   { Erroneous~variable~#1 used! }
10145 \msg_new:nnn { seq } { misused }
10146   { A~sequence~was~misused. }
10147 \msg_new:nnn { prg } { negative-replication }
10148   { Negative~argument~for~\iow_char:N\prg_replicate:nn. }
10149 \msg_new:nnn { prop } { prop-keyval }
10150   { Missing~'=~in~'#1'~(in~'...keyval:Nn') }
10151 \msg_new:nnn { kernel } { unknown-comparison }
10152   { Relation~'#1'~not~among~<,>,<=>,<=>,<=>. }
10153 \msg_new:nnn { kernel } { zero-step }
10154   { Zero~step~size~for~function~#1. }

```

Messages used by the “show” functions.

```

10155 \msg_new:nnn { clist } { show }
10156   {
10157     The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
10158     \tl_if_empty:nTF {#2}
10159     { is~empty \>~ . }
10160     { contains~the~items~(without~outer~braces): #2 . }
10161   }
10162 \msg_new:nnn { intarray } { show }
10163   { The~integer~array~#1~contains~#2~items: \> #3 . }
10164 \msg_new:nnn { prop } { show }
10165   {
10166     The~ \str_if_eq:nnF {#3} { flat } { #3~ }
10167     property~list~#1~
10168     \tl_if_empty:nTF {#2}
10169     { is~empty \>~ . }
10170     { contains~the~pairs~(without~outer~braces): #2 . }
10171   }
10172 \msg_new:nnn { seq } { show }
10173   {
10174     The~sequence~#1~
10175     \tl_if_empty:nTF {#2}
10176     { is~empty \>~ . }
10177     { contains~the~items~(without~outer~braces): #2 . }
10178   }
10179 \msg_new:nnn { kernel } { show-streams }
10180   {

```

```

10181 \tl_if_empty:nTF {#2} { No~ } { The~following~ }
10182 \str_case:nn {#1}
10183 {
10184   { ior } { input ~ }
10185   { iow } { output ~ }
10186 }
10187 streams~are~
10188 \tl_if_empty:nTF {#2} { open } { in~use: #2 . }
10189 }

System layer messages

10190 \msg_new:nnnn { sys } { backend-set }
10191 { Backend~configuration~already~set. }
10192 {
10193   Run~time~backend~selection~may~only~be~carried~out~once~during~a~run.~
10194   This~second~attempt~to~set~them~will~be~ignored.
10195 }
10196 \msg_new:nnnn { sys } { load-debug-in-preamble }
10197 { Load~debug~support~in~the~preamble. }
10198 {
10199   Debugging~requires~support~loaded~in~the~preamble: \\
10200   Use~\sys_load_debug:~before~\begin{document}.
10201 }
10202 \msg_new:nnnn { sys } { wrong-backend }
10203 { Backend~request~inconsistent~with~engine:~using~'~#2'~backend. }
10204 {
10205   You~have~requested~backend~'~#1'~,~but~this~is~not~suitable~for~use~with~the~
10206   active~engine.~LaTeX~will~use~the~'~#2'~backend~instead.
10207 }

```

53.8 Expandable errors

`\_msg_expandable_error:nn` In expansion only context, we cannot use the normal means of reporting errors. Instead, we rely on a low-level $\text{\TeX}$ error caused by expanding a macro `\???` with parameter text “?” (this could be any token) which we used followed by something else (here, a space). This shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \???
! mypkg Error: The error message.

```

In other words, $\text{\TeX}$ is processing the argument of `\use:n`, which is `\??? <space> ! <error type> : <error message>`.

```

10208 \cs_set_protected:Npn \_msg_tmp:w #1
10209 {
10210   \cs_new:Npn #1 ? { }
10211   \cs_new:Npn \_msg_expandable_error:nn ##1##2
10212   {
10213     \exp_after:wN \exp_after:wN
10214     \exp_after:wN \_msg_use_none_delimit_by_s_stop:w
10215     \use:n { #1 ~ ! ~ ##2 : ~ ##1 } \s_msg_stop
10216   }
10217 }
10218 \exp_args:Nc \_msg_tmp:w { ??? }

```

(End of definition for `\_msg_expandable_error:nn`.)

`\msg_expandable_error:nnnnnn` The command built from the csname `\c\_msg_text_prefix_tl #1 / #2` takes four arguments and builds the error text, which is fed to `\_msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded. The module name also has to be determined.

```

10219 \exp_args_generate:n { oooo }
10220 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
10221 {
10222   \exp_args:Nee \_msg_expandable_error:nn
10223   {
10224     \exp_args:Nc \exp_args:Noooo
10225     { \c\_msg_text_prefix_tl #1 / #2 }
10226     { \tl_to_str:n {#3} }
10227     { \tl_to_str:n {#4} }
10228     { \tl_to_str:n {#5} }
10229     { \tl_to_str:n {#6} }
10230   }
10231   { \msg_error_text:n {#1} }
10232 }
10233 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
10234 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
10235 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
10236 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
10237 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
10238 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
10239 \cs_new:Npn \msg_expandable_error:nn #1#2
10240 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
10241 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nneeee , nnffff }
10242 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nneee , nnfff }
10243 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnee , nnff }
10244 \cs_generate_variant:Nn \msg_expandable_error:nnn { nne , nnf }

```

(End of definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 91.)

53.9 Message formatting

```

10245 \prop_gput:Nnn \g_msg_module_name_prop { kernel } { LaTeX }
10246 \prop_gput:Nnn \g_msg_module_type_prop { kernel } { }
10247 \clist_map_inline:nn
10248 {
10249   char , clist , coffin , debug , deprecation , dim , msg ,
10250   quark , prg , prop , scanmark , seq , sys
10251 }
10252 {
10253   \prop_gput:Nnn \g_msg_module_name_prop {#1} { LaTeX }
10254   \prop_gput:Nnn \g_msg_module_type_prop {#1} { }
10255 }
10256 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / cmd } { LaTeX }
10257 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / cmd } { }
10258 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / ltcmd } { LaTeX }
10259 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / ltcmd } { }

```


10260 `</code>`

Chapter 54

l3file implementation

The following test files are used for this code: m3file001.

```
10261 <{*code>
```

54.1 Input operations

```
10262 <@@=ior>
```

54.1.1 Variables and constants

`\l__ior_tmp_tl` Used as a short-term scratch variable.

```
10263 \tl_new:N \l__ior_tmp_tl
```

(End of definition for \l__ior_tmp_tl.)

`\c__ior_term_ior` Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
10264 \int_const:Nn \c__ior_term_ior { 16 }
```

(End of definition for \c__ior_term_ior.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack.

```
10265 \seq_new:N \g__ior_streams_seq
```

(End of definition for \g__ior_streams_seq.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10266 \tl_new:N \l__ior_stream_tl
```

(End of definition for \l__ior_stream_tl.)

`\g__ior_streams_prop` The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For L^AT_EX 2_ε and plain T_EX this data is stored in `\count16`; with the `etex` package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConT_EXt, we need to look at `\count38` but there is no subtraction: like the original plain T_EX/L^AT_EX 2_ε mechanism it holds the value of the *last* stream allocated.

```
10267 \prop_new:N \g__ior_streams_prop
```

```

10268 \int_step_inline:nnn
10269 { 0 }
10270 {
10271   \cs_if_exist:NTF \contextversion
10272   { \tex_count:D 38 ~ }
10273   {
10274     \tex_count:D 16 ~ %
10275     \cs_if_exist:NT \loccount { - 1 }
10276   }
10277 }
10278 {
10279   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
10280 }

```

(End of definition for \g__ior_streams_prop.)

54.1.2 Stream management

\ior_new:N Reserving a new stream is done by defining the name as equal to using the terminal.
\ior_new:c

```

10281 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
10282 \cs_generate_variant:Nn \ior_new:N { c }

```

(End of definition for \ior_new:N. This function is documented on page 94.)

\g_tmpa_ior The usual scratch space.
\g_tmpb_ior

```

10283 \ior_new:N \g_tmpa_ior
10284 \ior_new:N \g_tmpb_ior

```

(End of definition for \g_tmpa_ior and \g_tmpb_ior. These variables are documented on page 102.)

\ior_open:Nn Use the conditional version, with an error if the file is not found.
\ior_open:cn

```

10285 \cs_new_protected:Npn \ior_open:Nn #1#2
10286 { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
10287 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End of definition for \ior_open:Nn. This function is documented on page 94.)

\l__ior_file_name_tl Data storage.

```

10288 \tl_new:N \l__ior_file_name_tl

```

(End of definition for \l__ior_file_name_tl.)

\ior_open:NnTF An auxiliary searches for the file in the T_EX, L^AT_EX 2_ε and L^AT_EX 3 paths. Then pass the
\ior_open:cnTF file found to the lower-level function which deals with streams. The full_name is empty
when the file is not found.

```

10289 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10290 {
10291   \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
10292   {
10293     \__kernel_ior_open:No #1 \l__ior_file_name_tl
10294     \prg_return_true:
10295   }
10296   { \prg_return_false: }
10297 }
10298 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

```

(End of definition for \ior_open:NnTF. This function is documented on page 94.)

`\__ior_new:N` Streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain `TeX`'s `\newread` being `\outer`. For `ConTeXt`, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```

10299 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10300   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10301 \cs_if_exist:NT \contextversion
10302   {
10303     \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
10304     \cs_gset_protected:Npn \__ior_new:N #1
10305       {
10306         \cs_undefine:N #1
10307         \__ior_new_aux:N #1
10308       }
10309   }

```

(End of definition for __ior_new:N.)

`\__kernel_ior_open:Nn` The stream allocation itself uses the fact that there is a list of all of those available.
`\__kernel_ior_open:No` Life gets more complex as it's important to keep things in sync. That is done using a
`\__ior_open_stream:Nn` two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain `TeX` or `LATeX 2ε` for a new stream and use that number (after a bit of conversion).

```

10310 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
10311   {
10312     \ior_close:N #1
10313     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10314     { \__ior_open_stream:Nn #1 {#2} }
10315     {
10316       \__ior_new:N #1
10317       \__kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10318       \__ior_open_stream:Nn #1 {#2}
10319     }
10320   }
10321 \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }

```

Here, we act defensively in case `LuaTeX` is in use with an extensionless file name.

```

10322 \cs_new_protected:Npe \__ior_open_stream:Nn #1#2
10323   {
10324     \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
10325     \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
10326     \tex_openin:D #1
10327     \sys_if_engine luatex:TF
10328       { {#2} }
10329       { \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
10330   }

```

(End of definition for __kernel_ior_open:Nn and __ior_open_stream:Nn.)

`\ior_shell_open:Nn` Actually much easier than either the standard `open` or `input` versions! When calling `\__ior_shell_open:nN` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `\__kernel_file_name_quote:n`.

```

10331 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
10332 {
10333   \sys_if_shell:TF
10334     { \__ior_shell_open:oN { \tl_to_str:n {#2} } #1 }
10335     { \msg_error:nn { kernel } { pipe-failed } }
10336 }
10337 \cs_new_protected:Npn \__ior_shell_open:nN #1#2
10338 {
10339   \tl_if_in:nnTF {#1} { " }
10340   {
10341     \msg_error:nne
10342       { kernel } { quote-in-shell } {#1}
10343   }
10344   { \__kernel_ior_open:Nn #2 { |#1 } }
10345 }
10346 \cs_generate_variant:Nn \__ior_shell_open:nN { o }
10347 \msg_new:nnnn { kernel } { pipe-failed }
10348 { Cannot~run~piped~system~commands. }
10349 {
10350   LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
10351   Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
10352 }

```

(End of definition for `\ior_shell_open:Nn` and `\__ior_shell_open:nN`. This function is documented on page 94.)

`\ior_close:N` Closing a stream means getting rid of it at the T_EX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

`\ior_close:c`

```

10353 \cs_new_protected:Npn \ior_close:N #1
10354 {
10355   \int_compare:nT { -1 < #1 < \c__ior_term_ior }
10356   {
10357     \tex_closein:D #1
10358     \prop_gremove:NV \g__ior_streams_prop #1
10359     \seq_if_in:NVF \g__ior_streams_seq #1
10360       { \seq_gpush:NV \g__ior_streams_seq #1 }
10361     \cs_gset_eq:NN #1 \c__ior_term_ior
10362   }
10363 }
10364 \cs_generate_variant:Nn \ior_close:N { c }

```

(End of definition for `\ior_close:N`. This function is documented on page 95.)

`\ior_show:N` Seek the stream in the `\g__ior_streams_prop` list, then show the stream as open or closed accordingly.

`\ior_log:N`

`\__ior_show:NN`

```

10365 \cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
10366 \cs_generate_variant:Nn \ior_show:N { c }
10367 \cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
10368 \cs_generate_variant:Nn \ior_log:N { c }

```

```

10369 \cs_new_protected:Npn \__ior_show:NN #1#2
10370 {
10371     \__kernel_chk_defined:NT #2
10372     {
10373         \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_tmp_tl
10374         {
10375             \exp_args:Ne #1
10376             { \token_to_str:N #2 ~ open: ~ \l__ior_tmp_tl }
10377         }
10378         { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
10379     }
10380 }

```

(End of definition for `\ior_show:N`, `\ior_log:N`, and `\__ior_show:NN`. These functions are documented on page 95.)

`\ior_show_list:` Show the property lists, but with some “pretty printing”. See the `l3msg` module. The
`\ior_log_list:` first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no
`\__ior_list:N` read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

10381 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nneeee }
10382 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nneeee }
10383 \cs_new_protected:Npn \__ior_list:N #1
10384 {
10385     #1 { kernel } { show-streams }
10386     { ior }
10387     {
10388         \prop_map_function:NN \g__ior_streams_prop
10389         \msg_show_item_unbraced:nn
10390     }
10391     { } { }
10392 }

```

(End of definition for `\ior_show_list:`, `\ior_log_list:`, and `\__ior_list:N`. These functions are documented on page 95.)

54.1.3 Reading input

`\if_eof:w` The primitive conditional

```

10393 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End of definition for `\if_eof:w`. This function is documented on page 102.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` The primitive test can only deal with numbers in the range `[0,15]` so we catch outliers (they are exhausted).

```

10394 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
10395 {
10396     \if_int_compare:w -1 < #1
10397     \if_int_compare:w #1 < \c__ior_term_ior
10398     \if_eof:w #1
10399         \prg_return_true:
10400     \else:

```

```

10401         \prg_return_false:
10402         \fi:
10403     \else:
10404         \prg_return_true:
10405         \fi:
10406     \else:
10407         \prg_return_true:
10408         \fi:
10409 }

```

(End of definition for `\ior_if_eof:NTF`. This function is documented on page 98.)

`\ior_get:NN` And here we read from files.

```

\__ior_get:NN 10410 \cs_new_protected:Npn \ior_get:NN #1#2
\ior_get:NNTF 10411 { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10412 \cs_new_protected:Npn \__ior_get:NN #1#2
10413 { \tex_read:D #1 to #2 }
10414 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
10415 {
10416     \ior_if_eof:NTF #1
10417     { \prg_return_false: }
10418     {
10419         \__ior_get:NN #1 #2
10420         \prg_return_true:
10421     }
10422 }

```

(End of definition for `\ior_get:NN`, `\__ior_get:NN`, and `\ior_get:NNTF`. These functions are documented on page 96.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline
`\__ior_str_get:NN` character and restore it afterwards.

```

\ior_str_get:NNTF 10423 \cs_new_protected:Npn \ior_str_get:NN #1#2
10424 { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10425 \cs_new_protected:Npn \__ior_str_get:NN #1#2
10426 {
10427     \exp_args:Nno \use:n
10428     {
10429         \int_set:Nn \tex_endlinechar:D { -1 }
10430         \tex_readline:D #1 to #2
10431         \int_set:Nn \tex_endlinechar:D
10432     } { \int_use:N \tex_endlinechar:D }
10433 }
10434 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
10435 {
10436     \ior_if_eof:NTF #1
10437     { \prg_return_false: }
10438     {
10439         \__ior_str_get:NN #1 #2
10440         \prg_return_true:
10441     }
10442 }

```

(End of definition for `\ior_str_get:NN`, `\__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 96.)

`\c__ior_term_noprompt_ior` For reading without a prompt.

```
10443 \int_const:Nn \c__ior_term_noprompt_ior { -1 }
```

(End of definition for `\c__ior_term_noprompt_ior`.)

`\ior_get_term:nN` Getting from the terminal is better with pretty-printing.

```
\ior_str_get_term:nN
\__ior_get_term:NnN
10444 \cs_new_protected:Npn \ior_get_term:nN #1#2
10445 { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
10446 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
10447 { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
10448 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
10449 {
10450   \group_begin:
10451   \tex_escapechar:D = -1 \scan_stop:
10452   \tl_if_blank:nTF {#2}
10453   { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
10454   { \exp_args:NNc #1 \c__ior_term_ior }
10455   {#2}
10456   \exp_args:NNNv \group_end:
10457   \tl_set:Nn #3 {#2}
10458 }
```

(End of definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `\__ior_get_term:NnN`. These functions are documented on page 99.)

`\ior_map_break:` Usual map breaking functions.

```
\ior_map_break:n
10459 \cs_new:Npn \ior_map_break:
10460 { \prg_map_break:Nn \ior_map_break: { } }
10461 \cs_new:Npn \ior_map_break:n
10462 { \prg_map_break:Nn \ior_map_break: }
```

(End of definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 98.)

`\ior_map_inline:Nn` Mapping over an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping cannot be nested with twice the same stream, as the stream has only one “current line”.

```
\ior_str_map_inline:Nn
\__ior_map_inline:NNn
\__ior_map_inline:NNNn
\__ior_map_inline_loop:NNN
10463 \cs_new_protected:Npn \ior_map_inline:Nn
10464 { \__ior_map_inline:NNn \__ior_get:NN }
10465 \cs_new_protected:Npn \ior_str_map_inline:Nn
10466 { \__ior_map_inline:NNn \__ior_str_get:NN }
10467 \cs_new_protected:Npn \__ior_map_inline:NNn
10468 {
10469   \int_gincr:N \g__kernel_prg_map_int
10470   \exp_args:Nc \__ior_map_inline:NNNn
10471   { \__ior_map_ \int_use:N \g__kernel_prg_map_int :n }
10472 }
10473 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
10474 {
10475   \cs_gset_protected:Npn #1 ##1 {#4}
10476   \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
10477   \prg_break_point:Nn \ior_map_break:
10478   { \int_gdecr:N \g__kernel_prg_map_int }
```



```

10479 }
10480 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
10481 {
10482     #2 #3 \l__ior_tmp_tl
10483     \if_eof:w #3
10484         \exp_after:wN \ior_map_break:
10485     \fi:
10486     \exp_args:No #1 \l__ior_tmp_tl
10487     \__ior_map_inline_loop:NNN #1#2#3
10488 }

```

(End of definition for `\ior_map_inline:Nn` and others. These functions are documented on page 97.)

```

\ior_map_variable:NNn
\ior_str_map_variable:NNn
\__ior_map_variable:NNNn
\__ior_map_variable_loop:NNNn

```

Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

```

10489 \cs_new_protected:Npn \ior_map_variable:NNn
10490 { \__ior_map_variable:NNNn \ior_get:NN }
10491 \cs_new_protected:Npn \ior_str_map_variable:NNn
10492 { \__ior_map_variable:NNNn \ior_str_get:NN }
10493 \cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
10494 {
10495     \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
10496     \prg_break_point:Nn \ior_map_break: { }
10497 }
10498 \cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
10499 {
10500     #1 #2 #3
10501     \if_eof:w #2
10502         \exp_after:wN \ior_map_break:
10503     \fi:
10504     #4
10505     \__ior_map_variable_loop:NNNn #1#2#3 {#4}
10506 }

```

(End of definition for `\ior_map_variable:NNn` and others. These functions are documented on page 97.)

54.2 Output operations

```

10507 <@@=iow>

```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

54.2.1 Variables and constants

`\l__iow_tmp_tl` Used as a short-term scratch variable.

```

10508 \tl_new:N \l__iow_tmp_tl

```

(End of definition for `\l__iow_tmp_tl`.)

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128

write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```

10509 \int_const:Nn \c_log_iow { -1 }
10510 \int_const:Nn \c_term_iow
10511 {
10512   \bool_lazy_and:nnTF
10513     { \sys_if_engine luatex_p: }
10514     { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
10515     { 128 }
10516     { 16 }
10517 }

```

(End of definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 102.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```

10518 \seq_new:N \g__iow_streams_seq

```

(End of definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```

10519 \tl_new:N \l__iow_stream_tl

```

(End of definition for `\l__iow_stream_tl`.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```

10520 \prop_new:N \g__iow_streams_prop
10521 \int_step_inline:nnn
10522 { 0 }
10523 {
10524   \cs_if_exist:NTF \contextversion
10525     { \tex_count:D 39 ~ }
10526     {
10527       \tex_count:D 17 ~
10528       \cs_if_exist:NT \loccount { - 1 }
10529     }
10530 }
10531 {
10532   \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by~format }
10533 }

```

(End of definition for `\g__iow_streams_prop`.)

54.2.2 Internal auxiliaries

`\s__iow_mark` Internal scan marks.

```

\s__iow_stop 10534 \scan_new:N \s__iow_mark
10535 \scan_new:N \s__iow_stop

```

(End of definition for `\s__iow_mark` and `\s__iow_stop`.)

`\__iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

10536 \cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}

```

(End of definition for `\__iow_use_i_delimit_by_s_stop:nw`.)

`\q__iow_nil` Internal quarks.
10537 `\quark_new:N \q__iow_nil`
(End of definition for \q__iow_nil.)

54.3 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:
`\iow_new:c` odd but at least consistent.

10538 `\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }`
10539 `\cs_generate_variant:Nn \iow_new:N { c }`

(End of definition for \iow_new:N. This function is documented on page 94.)

`\g_tmpa_iow` The usual scratch space.
`\g_tmpb_iow`
10540 `\iow_new:N \g_tmpa_iow`
10541 `\iow_new:N \g_tmpb_iow`

(End of definition for \g_tmpa_iow and \g_tmpb_iow. These variables are documented on page 102.)

`\__iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`. For ConTeXt, we have to deal with the fact that `\newwrite` works like our own: it actually checks before altering definition.

10542 `\exp_args:NNf \cs_new_protected:Npn \__iow_new:N`
10543 `{ \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }`
10544 `\cs_if_exist:NT \contextversion`
10545 `{`
10546 `\cs_new_eq:NN \__iow_new_aux:N \__iow_new:N`
10547 `\cs_gset_protected:Npn \__iow_new:N #1`
10548 `{`
10549 `\cs_undefine:N #1`
10550 `\__iow_new_aux:N #1`
10551 `}`
10552 `}`

(End of definition for __iow_new:N.)

`\l__iow_file_name_tl` Data storage.

10553 `\tl_new:N \l__iow_file_name_tl`

(End of definition for \l__iow_file_name_tl.)

`\iow_open:Nn` The same idea as for reading, but without the path and without the need to allow for a conditional version.
`\iow_open:NV`
`\iow_open:cn`
`\iow_open:cV`

10554 `\cs_new_protected:Npn \iow_open:Nn #1#2`
10555 `{`
10556 `\__kernel_tl_set:Nx \l__iow_file_name_tl`
10557 `{ \__kernel_file_name_sanitize:n {#2} }`
10558 `\__kernel_iow_open:No #1 \l__iow_file_name_tl`
10559 `}`
10560 `\cs_generate_variant:Nn \iow_open:Nn { NV , c , cV }`
10561 `\cs_new_protected:Npn \__kernel_iow_open:Nn #1#2`
10562 `{`

```

10563 \iow_close:N #1
10564 \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10565 { \__iow_open_stream:Nn #1 {#2} }
10566 {
10567   \__iow_new:N #1
10568   \__kernel_tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10569   \__iow_open_stream:Nn #1 {#2}
10570 }
10571 }
10572 \cs_generate_variant:Nn \__kernel_iow_open:Nn { No }
10573 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10574 {
10575   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10576   \prop_gput:Nvn \g__iow_streams_prop #1 {#2}
10577   \tex_immediate:D \tex_openout:D
10578     #1 \__kernel_file_name_quote:n {#2} \scan_stop:
10579 }
10580 \cs_generate_variant:Nn \__iow_open_stream:Nn { NV }

```

(End of definition for `\iow_open:Nn`, `\__kernel_iow_open:Nn`, and `\__iow_open_stream:Nn`. This function is documented on page 94.)

`\iow_shell_open:Nn`
`\__iow_shell_open:nN`
`\__iow_shell_open:oN`

Very similar to the `ior` version

```

10581 \cs_new_protected:Npn \iow_shell_open:Nn #1#2
10582 {
10583   \sys_if_shell:TF
10584     { \__iow_shell_open:oN { \tl_to_str:n {#2} } #1 }
10585     { \msg_error:nn { kernel } { pipe-failed } }
10586 }
10587 \cs_new_protected:Npn \__iow_shell_open:nN #1#2
10588 {
10589   \tl_if_in:nnTF {#1} { " }
10590   {
10591     \msg_error:nne
10592       { kernel } { quote-in-shell } {#1}
10593   }
10594   { \__kernel_iow_open:Nn #2 { |#1 } }
10595 }
10596 \cs_generate_variant:Nn \__iow_shell_open:nN { o }

```

(End of definition for `\iow_shell_open:Nn` and `\__iow_shell_open:nN`. This function is documented on page 94.)

`\iow_close:N`
`\iow_close:c`

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```

10597 \cs_new_protected:Npn \iow_close:N #1
10598 {
10599   \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
10600   {
10601     \tex_immediate:D \tex_closeout:D #1
10602     \prop_gremove:Nv \g__iow_streams_prop #1
10603     \seq_if_in:NvF \g__iow_streams_seq #1
10604       { \seq_gpush:Nv \g__iow_streams_seq #1 }
10605     \cs_gset_eq:NN #1 \c_term_iow

```

```

10606     }
10607   }
10608   \cs_generate_variant:Nn \iow_close:N { c }

```

(End of definition for `\iow_close:N`. This function is documented on page 95.)

```

\iow_show:N Seek the stream in the \g__iow_streams_prop list, then show the stream as open or
\iow_log:N closed accordingly.
\__iow_show:NN
10609   \cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
10610   \cs_generate_variant:Nn \iow_show:N { c }
10611   \cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
10612   \cs_generate_variant:Nn \iow_log:N { c }
10613   \cs_new_protected:Npn \__iow_show:NN #1#2
10614     {
10615       \__kernel_chk_defined:NT #2
10616       {
10617         \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_tmp_tl
10618         {
10619           \exp_args:Ne #1
10620             { \token_to_str:N #2 ~ open: ~ \l__iow_tmp_tl }
10621         }
10622         { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
10623       }
10624     }

```

(End of definition for `\iow_show:N`, `\iow_log:N`, and `\__iow_show:NN`. These functions are documented on page 95.)

```

\iow_show_list: Done as for input, but with a copy of the auxiliary so the name is correct.
\iow_log_list:
\__iow_list:N
10625   \cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nneeee }
10626   \cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nneeee }
10627   \cs_new_protected:Npn \__iow_list:N #1
10628     {
10629       #1 { kernel } { show-streams }
10630       { iow }
10631       {
10632         \prop_map_function:NN \g__iow_streams_prop
10633           \msg_show_item_unbraced:nn
10634       }
10635       { } { }
10636     }

```

(End of definition for `\iow_show_list:`, `\iow_log_list:`, and `\__iow_list:N`. These functions are documented on page 95.)

54.3.1 Deferred writing

```

\iow_shipout_e:Nn First the easy part, this is the primitive, which expects its argument to be braced.
\iow_shipout_e:Ne
\iow_shipout_e:cn
\iow_shipout_e:ce
10637   \cs_new_protected:Npn \iow_shipout_e:Nn #1#2
10638     { \tex_write:D #1 {#2} }
10639   \cs_generate_variant:Nn \iow_shipout_e:Nn { Ne , c, ce }

```

(End of definition for `\iow_shipout_e:Nn`. This function is documented on page 100.)

`\iow_shipout:Nn` With ε -TeX available deferred writing without expansion is easy.

`\iow_shipout:Ne` 10640 `\cs_new_protected:Npn \iow_shipout:Nn #1#2`

`\iow_shipout:Nx` 10641 `{ \tex_write:D #1 { \exp_not:n {#2} } }`

`\iow_shipout:cn` 10642 `\cs_generate_variant:Nn \iow_shipout:Nn { Ne , c , ce }`

`\iow_shipout:ce` 10643 `\cs_generate_variant:Nn \iow_shipout:Nn { Nx , cx }`

`\iow_shipout:cx` (End of definition for `\iow_shipout:Nn`. This function is documented on page 99.)

54.3.2 Immediate writing

`\__kernel_iow_with:Nnn` If the integer `#1` is equal to `#2`, just leave `#3` in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

`\__iow_with:nNnn`

`\__iow_with:oNnn`

10644 `\cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2`

10645 `{`

10646 `\int_compare:nNnTF {#1} = {#2}`

10647 `{ \use:n }`

10648 `{ \__iow_with:oNnn { \int_use:N #1 } #1 {#2} }`

10649 `}`

10650 `\cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4`

10651 `{`

10652 `\int_set:Nn #2 {#3}`

10653 `#4`

10654 `\int_set:Nn #2 {#1}`

10655 `}`

10656 `\cs_generate_variant:Nn \__iow_with:nNnn { o }`

(End of definition for `\__kernel_iow_with:Nnn` and `\__iow_with:nNnn`.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the `Nx` variant, because it differs in subtle ways from `x`-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `\__kernel_iow_with:Nnn` to support formats such as plain TeX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TeX looks at the value of the `\newlinechar` at shipout time in those cases.

`\iow_now:NV`

`\iow_now:Ne`

`\iow_now:Nx`

`\iow_now:cn`

`\iow_now:cV`

`\iow_now:ce`

`\iow_now:cx`

10657 `\cs_new_protected:Npn \iow_now:Nn #1#2`

10658 `{`

10659 `\__kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }`

10660 `{ \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }`

10661 `}`

10662 `\cs_generate_variant:Nn \iow_now:Nn { NV , Ne , c , cV , ce }`

10663 `\cs_generate_variant:Nn \iow_now:Nn { Nx , cx }`

(End of definition for `\iow_now:Nn`. This function is documented on page 99.)

`\iow_log:n` Writing to the log, the `\showstream`, and the terminal directly are relatively easy.

`\iow_log:e` 10664 `\cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }`

`\iow_log:x` 10665 `\cs_new_protected:Npn \iow_log:e { \iow_now:Ne \c_log_iow }`

`\iow_show:n` 10666 `\cs_generate_variant:Nn \iow_log:n { x }`

`\iow_show:e` 10667 `\cs_new_protected:Npn \iow_show:n { \iow_now:Nn \tex_showstream:D }`

`\iow_term:n`

`\iow_term:e`

`\iow_term:x`

```

10668 \cs_new_protected:Npn \iow_show:e { \iow_now:Ne \tex_showstream:D }
10669 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
10670 \cs_new_protected:Npn \iow_term:e { \iow_now:Ne \c_term_iow }
10671 \cs_generate_variant:Nn \iow_term:n { x }

```

(End of definition for `\iow_log:n`, `\iow_show:n`, and `\iow_term:n`. These functions are documented on page 99.)

For the above to work, we have to have a non-negative value for `\tex_showstream:D`, as `\tex_write:D` and `\tex_show:D/\tex_showtokens:D` treat negative values differently. This is needed for our approach to work at all, so we apply even when L^AT_EX is not in use. That doesn't work for LuaMetaT_EX, for the moment we simply skip. (For ConT_EXt we really need a different approach to showing messages anyway.)

```

10672 \cs_if_exist:NT \tex_showstream:D
10673 { \int_gset_eq:NN \tex_showstream:D \c_term_iow }

```

54.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```

10674 \cs_new:Npn \iow_newline: { ^^J }

```

(End of definition for `\iow_newline:.` This function is documented on page 100.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

10675 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End of definition for `\iow_char:N`. This function is documented on page 100.)

54.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by T_EX Live and MiK_T_EX.

```

10676 \int_new:N \l_iow_line_count_int
10677 \int_set:Nn \l_iow_line_count_int { 78 }

```

(End of definition for `\l_iow_line_count_int`. This variable is documented on page 102.)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the *<run-on text>*.

```

10678 \tl_new:N \l__iow_newline_tl

```

(End of definition for `\l__iow_newline_tl`.)

`\l__iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```

10679 \int_new:N \l__iow_line_target_int

```

(End of definition for `\l__iow_line_target_int`.)

`\__iow_set_indent:n` The `one_indent` variables hold one indentation marker and its length. The `\__iow_unindent:w` auxiliary removes one indentation. The function `\__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```

10680 \tl_new:N \l__iow_one_indent_tl
10681 \int_new:N \l__iow_one_indent_int
10682 \cs_new:Npn \__iow_unindent:w { }
10683 \cs_new_protected:Npn \__iow_set_indent:n #1
10684 {
10685   \__kernel_tl_set:Nx \l__iow_one_indent_tl
10686   { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
10687   \int_set:Nn \l__iow_one_indent_int
10688   { \str_count:N \l__iow_one_indent_tl }
10689   \exp_last_unbraced:NNo
10690   \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
10691 }
10692 \exp_args:Ne \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }

```

(End of definition for `\__iow_set_indent:n` and others.)

`\l__iow_indent_tl` The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of
`\l__iow_indent_int` characters.

```

10693 \tl_new:N \l__iow_indent_tl
10694 \int_new:N \l__iow_indent_int

```

(End of definition for `\l__iow_indent_tl` and `\l__iow_indent_int`.)

`\l__iow_line_tl` These hold the current line of text and a partial line to be added to it, respectively.
`\l__iow_line_part_tl`

```

10695 \tl_new:N \l__iow_line_tl
10696 \tl_new:N \l__iow_line_part_tl

```

(End of definition for `\l__iow_line_tl` and `\l__iow_line_part_tl`.)

`\l__iow_line_break_bool` Indicates whether the line was broken precisely at a chunk boundary.

```

10697 \bool_new:N \l__iow_line_break_bool

```

(End of definition for `\l__iow_line_break_bool`.)

`\l__iow_wrap_tl` Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```

10698 \tl_new:N \l__iow_wrap_tl

```

(End of definition for `\l__iow_wrap_tl`.)

`\c__iow_wrap_marker_tl` Every special action of the wrapping code is starts with the same recognizable string,
`\c__iow_wrap_end_marker_tl` `\c__iow_wrap_marker_tl`. Upon seeing that “word”, the wrapping code reads one space-
`\c__iow_wrap_newline_marker_tl` delimited argument to know what operation to perform. The setting of `\escapechar` here
`\c__iow_wrap_allow_break_marker_tl` is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```

10699 \group_begin:
10700   \int_set:Nn \tex_escapechar:D { -1 }
10701   \tl_const:Ne \c__iow_wrap_marker_tl
10702   { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
10703 \group_end:
10704 \tl_map_inline:nn

```



```

10705 { { end } { newline } { allow_break } { indent } { unindent } }
10706 {
10707     \tl_const:ce { c__iow_wrap_ #1 _marker_tl }
10708     {
10709         \c__iow_wrap_marker_tl
10710         #1
10711         \c_catcode_other_space_tl
10712     }
10713 }

```

(End of definition for `\c__iow_wrap_marker_tl` and others.)

`\iow_wrap_allow_break:` We set `\iow_wrap_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_wrap_allow_break:` when valid and otherwise to `\__iow_wrap_allow_break_error:`. The second produces an error expandably.

```

10714 \cs_new_protected:Npn \iow_wrap_allow_break:
10715 {
10716     \msg_error:nnnn { kernel } { iow-indent }
10717     { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
10718 }
10719 \cs_new:Npe \__iow_wrap_allow_break: { \c__iow_wrap_allow_break_marker_tl }
10720 \cs_new:Npn \__iow_wrap_allow_break_error:
10721 {
10722     \msg_expandable_error:nnnn { kernel } { iow-indent }
10723     { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
10724 }

```

(End of definition for `\iow_wrap_allow_break:`, `\__iow_wrap_allow_break:`, and `\__iow_wrap_allow_break_error:`. This function is documented on page 101.)

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `\__iow_indent:n` when valid and otherwise to `\__iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

10725 \cs_new_protected:Npn \iow_indent:n #1
10726 {
10727     \msg_error:nnnnn { kernel } { iow-indent }
10728     { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10729     #1
10730 }
10731 \cs_new:Npe \__iow_indent:n #1
10732 {
10733     \c__iow_wrap_indent_marker_tl
10734     #1
10735     \c__iow_wrap_unindent_marker_tl
10736 }
10737 \cs_new:Npn \__iow_indent_error:n #1
10738 {
10739     \msg_expandable_error:nnnnn { kernel } { iow-indent }
10740     { \iow_wrap:nnnN } { \iow_indent:n } {#1}
10741     #1
10742 }

```

(End of definition for `\iow_indent:n`, `\__iow_indent:n`, and `\__iow_indent_error:n`. This function is documented on page 101.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `\_` and other formatting commands the correct definition for messages and perform the given setup #3. `\iow_wrap:nenN` The definition of `\_` uses an “other” space rather than a normal space, because the latter might be absorbed by $\TeX$ to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```

10743 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
10744 {
10745   \group_begin:
10746   \cs_if_exist_use:N \conditionally@traceoff
10747   \int_set:Nn \tex_escapechar:D { -1 }
10748   \cs_set:Npe \{ { \token_to_str:N \{ }
10749   \cs_set:Npe \# { \token_to_str:N \# }
10750   \cs_set:Npe \} { \token_to_str:N \} }
10751   \cs_set:Npe \% { \token_to_str:N \% }
10752   \cs_set:Npe \~ { \token_to_str:N \~ }
10753   \int_set:Nn \tex_escapechar:D { 92 }
10754   \cs_set_eq:NN \ \ \iow_newline:
10755   \cs_set_eq:NN \ \_ \c_catcode_other_space_tl
10756   \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break:
10757   \cs_set_eq:NN \iow_indent:n \__iow_indent:n
10758   #3

```

Then fully-expand the input: in package mode, the expansion uses $\text{\LaTeX 2}_\epsilon$ ’s `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

10759   \cs_set_eq:NN \protect \token_to_str:N
10760   \__kernel_tl_set:Nx \l__iow_wrap_tl {#1}
10761   \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break_error:
10762   \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```

10763   \__kernel_tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
10764   \__kernel_tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
10765   \int_set:Nn \l__iow_line_target_int
10766   { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

Sanity check.

```

10767   \int_compare:nNnT { \l__iow_line_target_int } < 0
10768   {
10769     \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
10770     \int_set:Nn \l__iow_line_target_int
10771     { \l_iow_line_count_int + 1 }
10772   }

```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l__iow_wrap_tl`.

```

10773     \__iow_wrap_do:
10774     \exp_args:NNf \group_end:
10775     #4 { \tl_to_str:N \l__iow_wrap_tl }
10776   }
10777 \cs_generate_variant:Nn \iow_wrap:nnnN { ne }

```

(End of definition for \iow_wrap:nnnN. This function is documented on page 101.)

__iow_wrap_do: Escape spaces and change newlines to \c__iow_wrap_newline_marker_tl. Set up a few variables, in particular the initial value of \l__iow_wrap_tl: the space stops the f-expansion of the main wrapping function and \use_none:n removes a newline marker inserted by later code. The main loop consists of repeatedly calling the chunk auxiliary to wrap chunks delimited by (newline or indentation) markers.

```

10778 \cs_new_protected:Npn \__iow_wrap_do:
10779 {
10780   \__kernel_tl_set:Nx \l__iow_wrap_tl
10781   {
10782     \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
10783     \c__iow_wrap_end_marker_tl
10784   }
10785   \__kernel_tl_set:Nx \l__iow_wrap_tl
10786   {
10787     \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
10788     ^^J \q__iow_nil ^^J \s__iow_stop
10789   }
10790   \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
10791 }
10792 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
10793 {
10794   #1
10795   \if_meaning:w \q__iow_nil #2
10796   \__iow_use_i_delimit_by_s_stop:nw
10797   \fi:
10798   \c__iow_wrap_newline_marker_tl
10799   \__iow_wrap_fix_newline:w #2 ^^J
10800 }
10801 \cs_new_protected:Npn \__iow_wrap_start:w
10802 {
10803   \bool_set_false:N \l__iow_line_break_bool
10804   \tl_clear:N \l__iow_line_tl
10805   \tl_clear:N \l__iow_line_part_tl
10806   \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
10807   \int_zero:N \l__iow_indent_int
10808   \tl_clear:N \l__iow_indent_tl
10809   \__iow_wrap_chunk:nw { \l__iow_line_count_int }
10810 }

```

(End of definition for __iow_wrap_do:, __iow_wrap_fix_newline:w, and __iow_wrap_start:w.)

__iow_sep:

```

10811 \cs_new_eq:NN \__iow_sep: \__kernel_int_sep:

```

(End of definition for __iow_sep:.)

`\__iow_wrap_chunk:nw` The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `\__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (`#1`) before the `\__iow_wrap_end_chunk:w` auxiliary.

```

10812 \cs_set_protected:Npn \__iow_tmp:w #1#2
10813 {
10814   \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
10815   {
10816     \tl_if_empty:NTF {##2}
10817     {
10818       \tl_clear:N \l__iow_line_part_tl
10819       \__iow_wrap_next:nw {##1}
10820     }
10821     {
10822       \tl_if_empty:NTF \l__iow_line_tl
10823       {
10824         \__iow_wrap_line:nw
10825         { \l__iow_indent_tl }
10826         ##1 - \l__iow_indent_int \__iow_sep:
10827       }
10828       { \__iow_wrap_line:nw { } ##1 \__iow_sep: }
10829       ##2 #1
10830       \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
10831     }
10832   }
10833   \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
10834   { \use:c { __iow_wrap_##2:n } {##1} }
10835 }
10836 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl

```

(End of definition for `\__iow_wrap_chunk:nw` and `\__iow_wrap_next:nw`.)

`\__iow_wrap_line:nw` This is followed by `{⟨string⟩⟨int expr⟩\__iow_sep:}`. It stores the `⟨string⟩` and up to `⟨int expr⟩` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. `\__iow_wrap_line_loop:w` When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k , then $7-k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments `#2–#9` of the `line_loop` auxiliary or as one of the arguments `#2–#8` of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop

the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```

10837 \cs_new_protected:Npn \__iow_wrap_line:nw #1
10838 {
10839   \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
10840   #1
10841   \exp_after:wN \__iow_wrap_line_loop:w
10842   \int_value:w \int_eval:w
10843 }
10844 \cs_new:Npn \__iow_wrap_line_loop:w #1 \__iow_sep: #2#3#4#5#6#7#8#9
10845 {
10846   \if_int_compare:w #1 < 8 \exp_stop_f:
10847     \__iow_wrap_line_aux:Nw #1
10848   \fi:
10849   #2 #3 #4 #5 #6 #7 #8 #9
10850   \exp_after:wN \__iow_wrap_line_loop:w
10851   \int_value:w \int_eval:w #1 - 8 \__iow_sep:
10852 }
10853 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 \__iow_sep:
10854 {
10855   #2
10856   \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
10857   \exp_after:wN #1
10858   \exp:w \exp_end_continue_f:w
10859   \exp_after:wN \exp_after:wN
10860   \if_case:w #1 \exp_stop_f:
10861     \prg_do_nothing:
10862   \or: \use_none:n
10863   \or: \use_none:nn
10864   \or: \use_none:nnn
10865   \or: \use_none:nnnn
10866   \or: \use_none:nnnnn
10867   \or: \use_none:nnnnnn
10868   \or: \__iow_wrap_line_seven:nnnnnnn
10869   \fi:
10870   { } { } { } { } { } { } { } { } { } #3
10871 }
10872 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
10873 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
10874 {
10875   #2 #3 #4 #5 #6 #7 #8
10876   \use_none:nnnnn \int_eval:w 8 - \__iow_sep: #9
10877   \token_if_eq_charcode:NNTF \c_space_token #9
10878     { \__iow_wrap_line_end:nw { } }
10879     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
10880 }
10881 \cs_new:Npn \__iow_wrap_line_end:nw #1
10882 {
10883   \if_false: { \fi: }
10884   \__iow_wrap_store_do:n {#1}
10885   \__iow_wrap_next_line:w
10886 }
10887 \cs_new:Npn \__iow_wrap_end_chunk:w
10888   #1 \int_eval:w #2 - #3 \__iow_sep: #4#5 \s__iow_stop

```

```

10889 {
10890   \if_false: { \fi: }
10891   \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
10892 }

```

(End of definition for __iow_wrap_line:nw and others.)

__iow_wrap_break:w Functions here are defined indirectly: __iow_tmp:w is eventually called with an “other” space as its argument. The goal is to remove from \l__iow_line_part_tl the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument `##3` is ? __iow_wrap_break_end:w instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls __iow_wrap_line_end:nw to finish up the line and move on to the next. If there is no space in \l__iow_line_part_tl then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including `##4`, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```

10893 \cs_set_protected:Npn \__iow_tmp:w #1
10894 {
10895   \cs_new:Npn \__iow_wrap_break:w
10896   {
10897     \tex_edef:D \l__iow_line_part_tl
10898     { \if_false: } \fi:
10899     \exp_after:wN \__iow_wrap_break_first:w
10900     \l__iow_line_part_tl
10901     #1
10902     { ? \__iow_wrap_break_end:w }
10903     \s__iow_mark
10904   }
10905   \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
10906   {
10907     \use_none:nn ##2 \__iow_wrap_break_none:w
10908     \__iow_wrap_break_loop:w ##1 #1 ##2
10909   }
10910   \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
10911   {
10912     \tl_if_empty:NTF \l__iow_line_tl
10913     { ##2 ##4 \__iow_wrap_line_end:nw { } }
10914     { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
10915   }
10916   \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
10917   {
10918     \use_none:n ##3
10919     ##1 #1
10920     \__iow_wrap_break_loop:w ##2 #1 ##3
10921   }
10922   \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
10923   { ##1 \__iow_wrap_line_end:nw { } ##3 }
10924 }
10925 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End of definition for __iow_wrap_break:w and others.)

`\__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

10926 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
10927 {
10928   \tl_clear:N \l__iow_line_tl
10929   \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
10930   {
10931     \tl_clear:N \l__iow_line_part_tl
10932     \bool_set_true:N \l__iow_line_break_bool
10933     \__iow_wrap_next:nw { \l__iow_line_target_int }
10934   }
10935   {
10936     \__iow_wrap_line:nw
10937     { \l__iow_indent_tl }
10938     \l__iow_line_target_int - \l__iow_indent_int \__iow_sep:
10939     #1 #2 \s__iow_stop
10940   }
10941 }

```

(End of definition for `\__iow_wrap_next_line:w`.)

`\__iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow-break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

10942 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
10943 {
10944   \__kernel_tl_set:Nx \l__iow_line_tl
10945   { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
10946   \bool_set_false:N \l__iow_line_break_bool
10947   \tl_if_empty:NNTF \l__iow_line_part_tl
10948   { \__iow_wrap_chunk:nw {#1} }
10949   { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
10950 }

```

(End of definition for `\__iow_wrap_allow_break:n`.)

`\__iow_wrap_indent:n` `\__iow_wrap_unindent:n` These functions are called after a chunk has been wrapped, when encountering `indent/unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

10951 \cs_new_protected:Npn \__iow_wrap_indent:n #1
10952 {
10953   \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
10954   \bool_set_false:N \l__iow_line_break_bool
10955   \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10956   \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
10957   \__iow_wrap_chunk:nw {#1}
10958 }
10959 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
10960 {

```

```

10961 \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
10962 \bool_set_false:N \l__iow_line_break_bool
10963 \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10964 \__kernel_tl_set:Nx \l__iow_indent_tl
10965 { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
10966 \__iow_wrap_chunk:nw {#1}
10967 }

```

(End of definition for __iow_wrap_indent:n and __iow_wrap_unindent:n.)

__iow_wrap_newline:n These functions are called after a chunk has been line-wrapped, when encountering a
 __iow_wrap_end:n **newline/end** marker. Unless we just took a line-break, store the line part and the line
 so far into the whole \l__iow_wrap_tl, trimming a trailing space. In the **newline** case
 look for a new line (of length \l__iow_line_target_int) in a new chunk.

```

10968 \cs_new_protected:Npn \__iow_wrap_newline:n #1
10969 {
10970   \bool_if:NF \l__iow_line_break_bool
10971   { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10972   \bool_set_false:N \l__iow_line_break_bool
10973   \__iow_wrap_chunk:nw { \l__iow_line_target_int }
10974 }
10975 \cs_new_protected:Npn \__iow_wrap_end:n #1
10976 {
10977   \bool_if:NF \l__iow_line_break_bool
10978   { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10979   \bool_set_false:N \l__iow_line_break_bool
10980 }

```

(End of definition for __iow_wrap_newline:n and __iow_wrap_end:n.)

__iow_wrap_store_do:n First add the last line part to the line, then append it to \l__iow_wrap_tl with the
 appropriate new line (with “run-on” text), possibly with its last space removed (#1 is
 empty or __iow_wrap_trim:N).

```

10981 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
10982 {
10983   \__kernel_tl_set:Nx \l__iow_line_tl
10984   { \l__iow_line_tl \l__iow_line_part_tl }
10985   \__kernel_tl_set:Nx \l__iow_wrap_tl
10986   {
10987     \l__iow_wrap_tl
10988     \l__iow_newline_tl
10989     #1 \l__iow_line_tl
10990   }
10991   \tl_clear:N \l__iow_line_tl
10992 }

```

(End of definition for __iow_wrap_store_do:n.)

__iow_wrap_trim:N Remove one trailing “other” space from the argument if present.

```

\__iow_wrap_trim:w
\__iow_wrap_trim_aux:w
10993 \cs_set_protected:Npn \__iow_tmp:w #1
10994 {
10995   \cs_new:Npn \__iow_wrap_trim:N ##1
10996   { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
10997   \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark

```



```

10998     { \_iow_wrap_trim_aux:w ##1 \s\_iow\_mark }
10999     \cs\_new:Npn \_iow\_wrap\_trim\_aux:w ##1 \s\_iow\_mark ##2 \s\_iow\_stop {##1}
11000   }
11001   \exp\_args:Nv \_iow\_tmp:w \c\_catcode\_other\_space\_tl
(End of definition for \_iow\_wrap\_trim:N, \_iow\_wrap\_trim:w, and \_iow\_wrap\_trim\_aux:w.)
11002   <@@=file>

```

54.4 File operations

`\l\_file\_tmp\_tl` Used as a short-term scratch variable.

```
11003 \tl\_new:N \l\_file\_tmp\_tl
```

(End of definition for `\l\_file\_tmp\_tl`.)

`\g\_file\_curr\_dir\_str`
`\g\_file\_curr\_ext\_str`
`\g\_file\_curr\_name\_str` The name of the current file should be available at all times: the name itself is set dynamically.

```

11004 \str\_new:N \g\_file\_curr\_dir\_str
11005 \str\_new:N \g\_file\_curr\_ext\_str
11006 \str\_new:N \g\_file\_curr\_name\_str

```

(End of definition for `\g\_file\_curr\_dir\_str`, `\g\_file\_curr\_ext\_str`, and `\g\_file\_curr\_name\_str`. These variables are documented on page 102.)

`\g\_file\_stack\_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by $\text{\LaTeX} 2_{\epsilon}$ (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As $\text{\LaTeX} 2_{\epsilon}$ doesn't store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```

11007 \seq\_new:N \g\_file\_stack\_seq
11008 \group\_begin:
11009   \cs\_set\_protected:Npn \_file\_tmp:w #1#2#3
11010   {
11011     \tl\_if\_blank:nTF {#1}
11012     {
11013       \cs\_set:Npn \_file\_tmp:w ##1 " ##2 " ##3 \s\_file\_stop
11014       { { } {##2} { } }
11015       \seq\_gput\_right:Ne \g\_file\_stack\_seq
11016       {
11017         \exp\_after:wN \_file\_tmp:w \tex\_jobname:D
11018         " \tex\_jobname:D " \s\_file\_stop
11019       }
11020     }
11021     {
11022       \seq\_gput\_right:Nn \g\_file\_stack\_seq { { } {#1} {#2} }
11023       \_file\_tmp:w
11024     }
11025   }
11026   \cs\_if\_exist:NT \@currnamestack
11027   {
11028     \tl\_if\_empty:NF \@currnamestack
11029     { \exp\_after:wN \_file\_tmp:w \@currnamestack }
11030   }
11031 \group\_end:

```

(End of definition for `\g__file_stack_seq`.)

`\g__file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

11032 `\seq_new:N \g__file_record_seq`

(End of definition for `\g__file_record_seq`.)

`\l__file_base_name_tl` For storing the basename and full path whilst passing data internally.

`\l__file_full_name_tl` 11033 `\tl_new:N \l__file_base_name_tl`

11034 `\tl_new:N \l__file_full_name_tl`

(End of definition for `\l__file_base_name_tl` and `\l__file_full_name_tl`.)

`\l__file_dir_str` Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

`\l__file_ext_str`

`\l__file_name_str`

11035 `\str_new:N \l__file_dir_str`

11036 `\str_new:N \l__file_ext_str`

11037 `\str_new:N \l__file_name_str`

(End of definition for `\l__file_dir_str`, `\l__file_ext_str`, and `\l__file_name_str`.)

`\l_file_search_path_seq` The current search path.

11038 `\seq_new:N \l_file_search_path_seq`

(End of definition for `\l_file_search_path_seq`. This variable is documented on page 103.)

`\l__file_tmp_seq` Scratch space for comma list conversion.

11039 `\seq_new:N \l__file_tmp_seq`

(End of definition for `\l__file_tmp_seq`.)

54.4.1 Internal auxiliaries

`\s__file_stop` Internal scan marks.

11040 `\scan_new:N \s__file_stop`

(End of definition for `\s__file_stop`.)

`\q__file_nil` Internal quarks.

11041 `\quark_new:N \q__file_nil`

(End of definition for `\q__file_nil`.)

`\__file_quark_if_nil_p:n` Branching quark conditional.

`\__file_quark_if_nil:nTF`

11042 `\__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }`

(End of definition for `\__file_quark_if_nil:nTF`.)

`\q__file_recursion_tail` Internal recursion quarks.

`\q__file_recursion_stop`

11043 `\quark_new:N \q__file_recursion_tail`

11044 `\quark_new:N \q__file_recursion_stop`

(End of definition for `\q__file_recursion_tail` and `\q__file_recursion_stop`.)

```

\__file_if_recursion_tail_break:NN
\__file_if_recursion_tail_stop_do:Nn

```

Functions to query recursion quarks.

```

11045 \__kernel_quark_new_test:N \__file_if_recursion_tail_stop:N
11046 \__kernel_quark_new_test:N \__file_if_recursion_tail_stop_do:nn

```

(End of definition for __file_if_recursion_tail_break:NN and __file_if_recursion_tail_stop_do:Nn.)

```

\__kernel_file_name_sanitiz:n
\__file_name_expand:n
\__file_name_expand_cleanup:Nw
\__file_name_expand_cleanup:w
\__file_name_expand_end:
\__file_name_expand_error:Nw
\__file_name_expand_error_aux:Nw
\__file_name_strip_quotes:n
\__file_name_strip_quotes:nnnw
\__file_name_strip_quotes:nnn
\__file_name_trim_spaces:n
\__file_name_trim_spaces:nw
\__file_name_trim_spaces_aux:n
\__file_name_trim_spaces_aux:w

```

Expanding the file name uses a \csname-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using \ifcsname. This is less conservative than the token-by-token approach used before, but it is much faster.

```

11047 \cs_new:Npn \__kernel_file_name_sanitiz:n #1
11048 {
11049   \exp_args:Ne \__file_name_trim_spaces:n
11050   {
11051     \exp_args:Ne \__file_name_strip_quotes:n
11052     { \__file_name_expand:n {#1} }
11053   }
11054 }

```

We'll use \cs:w to start expanding the file name, and to avoid creating csnames equal to \relax with “common” names, there's a prefix __file_name= to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```

11055 \cs_new:Npn \__file_name_expand:n #1
11056 {
11057   \exp_after:wN \__file_name_expand_cleanup:Nw
11058   \cs:w __file_name = #1 \cs_end:
11059   \__file_name_expand_end:
11060 }

```

With the csname built, we grab it, and grab the remaining tokens delimited by __file_name_expand_end:. If there are any remaining tokens, something bad happened, so we'll call the error procedure __file_name_expand_error:Nw. If everything went according to plan, then use \token_to_str:N on the csname built, and call __file_name_expand_cleanup:w to remove the prefix we added a while back. __file_name_expand_cleanup:w takes a leading argument so we don't have to bother about the value of \tex_escapechar:D.

```

11061 \cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
11062 {
11063   \tl_if_empty:nF {#2}
11064   { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
11065   \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
11066 }
11067 \exp_last_unbraced:NNNNo
11068 \cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }

```

In non-error cases __file_name_expand_end: should not expand. It will only do so in case there is a \csname too much in the file name, so it will throw an error (while expanding), then insert the missing \cs_end: and yet another __file_name_expand_end: that will be used as a delimiter by __file_name_expand_cleanup:Nw (or that will expand again if yet another \endcsname is missing).

```

11069 \cs_new:Npn \__file_name_expand_end:
11070 {

```

```

11071 \msg_expandable_error:nn
11072 { kernel } { filename-missing-endcsname }
11073 \cs_end: \__file_name_expand_end:
11074 }

```

Now to the error case. `\__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `\__file_name_expand_error_aux:Nw` throws the error.

```

11075 \cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
11076 { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
11077 \cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
11078 \__file_name_expand_end:
11079 {
11080 \msg_expandable_error:nnff
11081 { kernel } { filename-chars-lost }
11082 { \token_to_str:N #1 } { \exp_stop_f: #2 }
11083 }

```

Quoting file name uses basically the same approach as for `luaquotejobname:` count the " tokens and remove them.

```

11084 \cs_new:Npn \__file_name_strip_quotes:n #1
11085 {
11086 \__file_name_strip_quotes:nw { 0 }
11087 #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
11088 }
11089 \cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
11090 {
11091 \if_meaning:w \q__file_recursion_tail #2
11092 \__file_name_strip_quotes_end:wnwn
11093 \fi:
11094 #2
11095 \__file_name_strip_quotes:nw { #1 + 1 }
11096 }
11097 \cs_new:Npn \__file_name_strip_quotes_end:wnwn \fi: #1
11098 \__file_name_strip_quotes:nw #2 \q__file_recursion_stop #3
11099 {
11100 \fi:
11101 \int_if_odd:nT {#2}
11102 {
11103 \msg_expandable_error:nnn
11104 { kernel } { unbalanced-quote-in-filename } {#3}
11105 }
11106 }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

11107 \cs_new:Npn \__file_name_trim_spaces:n #1
11108 { \__file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
11109 \cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
11110 {
11111 \__file_quark_if_nil:nTF {#3}
11112 {
11113 \tl_trim_spaces_apply:nN { #1 \s__file_stop }

```

```

11114         \_file_name_trim_spaces_aux:n
11115     }
11116     { \tl_trim_spaces:n {#1} }
11117 }
11118 \cs_new:Npn \_file_name_trim_spaces_aux:n #1
11119 { \_file_name_trim_spaces_aux:w #1 }
11120 \cs_new:Npn \_file_name_trim_spaces_aux:w #1 \s__file_stop {#1}

```

(End of definition for _kernel_file_name_sanitize:n and others.)

```

\_kernel_file_name_quote:n
\_file_name_quote:nw
11121 \cs_new:Npn \_kernel_file_name_quote:n #1
11122 { \_file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
11123 \cs_new:Npn \_file_name_quote:nw #1 #2 ~ #3 \s__file_stop
11124 {
11125     \_file_quark_if_nil:nTF {#3}
11126     { #1 }
11127     { "#1" }
11128 }

```

(End of definition for _kernel_file_name_quote:n and _file_name_quote:nw.)

\c__file_marker_tl The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```

11129 \tl_const:Nc \c__file_marker_tl { : \token_to_str:N : }

```

(End of definition for \c__file_marker_tl.)

\file_get:nnN **\file_get:VnN** **\file_get:nnN** The approach here is similar to that for \tl_set_rescan:Nnn. The file contents are grabbed as an argument delimited by \c__file_marker_tl. A few subtleties: braces in \if_false: ... \fi: to deal with possible alignment tabs, \tracingnesting to avoid a warning about a group being closed inside the \scantokens, and \prg_return_true: is placed after the end-of-file marker.

```

11130 \cs_new_protected:Npn \file_get:nnN #1#2#3
11131 {
11132     \file_get:nnNF {#1} {#2} #3
11133     { \tl_set:Nn #3 { \q_no_value } }
11134 }
11135 \cs_generate_variant:Nn \file_get:nnN { V }
11136 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
11137 {
11138     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11139     {
11140         \exp_args:NV \_file_get_aux:nnN
11141         \l__file_full_name_tl
11142         {#2} #3
11143         \prg_return_true:
11144     }
11145     { \prg_return_false: }
11146 }
11147 \prg_generate_conditional_variant:Nnn \file_get:nnN { V } { T , F , TF }
11148 \cs_new_protected:Npe \_file_get_aux:nnN #1#2#3
11149 {
11150     \exp_not:N \if_false: { \exp_not:N \fi:

```

```

11151 \group_begin:
11152 \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
11153 \exp_not:N \exp_args:No \tex_everyleaf:D
11154 { \exp_not:N \c__file_marker_tl }
11155 #2 \scan_stop:
11156 \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
11157 \exp_not:N \exp_after:wN #3
11158 \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
11159 \exp_not:N \tex_input:D
11160 \sys_if_engine_luatex:TF
11161 { {#1} }
11162 { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11163 \exp_not:N \if_false: } \exp_not:N \fi:
11164 }
11165 \exp_args:Nno \use:nn
11166 { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
11167 { \c__file_marker_tl }
11168 {
11169 \group_end:
11170 \tl_set:No #1 {#2}
11171 }

```

(End of definition for `\file_get:nnNTF` and others. These functions are documented on page 106.)

`\__file_size:n` A copy of the primitive where it's available.

```

11172 \cs_new_eq:NN \__file_size:n \tex_filesize:D

```

(End of definition for `\__file_size:n`.)

`\file_full_name:n`

File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

\__file_full_name:n
\__file_full_name_aux:n
\__file_full_name_auxi:nn
\__file_full_name_auxii:nn
\__file_full_name_aux:Nnn
\__file_full_name_slash:n
\__file_full_name_slash:w
\__file_full_name_aux:nN
\__file_full_name_aux:nnN
\__file_name_cleanup:w
\__file_name_end:
\__file_name_ext_check:nn
\__file_name_ext_check:nnw
\__file_name_ext_check:nnnw
\__file_name_ext_check:nnnn
\__file_name_ext_check:nnnn
11173 \cs_new:Npn \file_full_name:n #1
11174 {
11175 \exp_args:Ne \__file_full_name:n
11176 { \__kernel_file_name_sanitize:n {#1} }
11177 }
11178 \cs_generate_variant:Nn \file_full_name:n { V }

```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of `\pdffilesize` is kept available as an argument. For package mode, `\input@path` is a token list not a sequence.

```

11179 \cs_new:Npn \__file_full_name:n #1
11180 {
11181 \tl_if_blank:nF {#1}
11182 { \exp_args:Nne \__file_full_name_auxii:nn {#1} { \__file_full_name_aux:n {#1} } }
11183 }

```

To avoid repeated reading of files we need to cache the loading: this is important as the code here is used by *all* file checks. The same marker is used in the L^AT_EX 2_ε kernel, meaning that we get a double-saving with for example `\IfFileExists`. As this is all

about performance, we use the low-level approach for the conditionals. For a file already seen, the size is reported as -1 so it's distinct from any non-cached ones.

```

11184 \cs_new:Npn \__file_full_name_aux:n #1
11185 {
11186   \if_cs_exist:w __file_seen_ \tl_to_str:n {#1} : \cs_end:
11187   -1
11188   \else:
11189     \exp_args:Ne \__file_full_name_auxi:nn { \__file_size:n {#1} } {#1}
11190   \fi:
11191 }

```

We will need the size of files later, and we have to avoid the `\scan_stop:` causing issues if we are raising the flag. Thus there is a slightly odd gobble here.

```

11192 \cs_new:Npn \__file_full_name_auxi:nn #1#2
11193 {
11194   \if:w \scan_stop: #1 \scan_stop:
11195   \else:
11196     \exp_after:wN \use_none:n
11197     \cs:w __file_seen_ \tl_to_str:n {#2} : \cs_end:
11198     #1
11199   \fi:
11200 }
11201 \cs_new:Npn \__file_full_name_auxii:nn #1 #2
11202 {
11203   \tl_if_blank:nTF {#2}
11204   {
11205     \seq_map_tokens:Nn \l_file_search_path_seq
11206     { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
11207     \cs_if_exist:NT \input@path
11208     {
11209       \tl_map_tokens:Nn \input@path
11210       { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
11211     }
11212     \__file_name_end:
11213   }
11214   { \__file_ext_check:nn {#1} {#2} }
11215 }

```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

11216 \cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
11217 {
11218   \exp_args:Ne \__file_full_name_aux:nN
11219   { \__file_full_name_slash:n {#3} #2 }
11220   #1
11221 }
11222 \cs_new:Npn \__file_full_name_slash:n #1
11223 {
11224   \__file_full_name_slash:nw {#1} #1 \q_nil / \q_nil / \q_nil \q_stop
11225 }
11226 \cs_new:Npn \__file_full_name_slash:nw #1#2 / \q_nil / #3 \q_stop
11227 {
11228   \quark_if_nil:nTF {#3}
11229   { #1 / }

```

```

11230     { #2 / }
11231   }
11232   \cs_new:Npn \__file_full_name_aux:nN #1
11233     { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_full_name_aux:n {#1} } }
11234   \cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
11235     {
11236       \tl_if_blank:nF {#2}
11237       {
11238         #3
11239         {
11240           \__file_ext_check:nn {#1} {#2}
11241           \__file_name_cleanup:w
11242         }
11243       }
11244     }
11245   \cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
11246   \cs_new:Npn \__file_name_end: { }

```

As $\text{\TeX}$ automatically adds `.tex` if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

11247   \cs_new:Npn \__file_ext_check:nn #1 #2
11248     { \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
11249   \cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
11250     {
11251       \__file_quark_if_nil:nTF {#4}
11252       {
11253         \exp_args:No \__file_ext_check:nnnw
11254         { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
11255       }
11256       { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
11257     }
11258   \cs_new:Npe \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
11259     {
11260       \exp_not:N \__file_quark_if_nil:nTF {#5}
11261       {
11262         \exp_not:N \__file_ext_check:nnn
11263         { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
11264       }
11265       { #1 #3 }
11266     }
11267   \cs_new:Npn \__file_ext_check:nnn #1
11268     { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_full_name_aux:n {#1} } }
11269   \cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
11270     {
11271       \tl_if_blank:nTF {#2}
11272       {#3}
11273       {
11274         \bool_lazy_or:nnTF
11275         { \int_compare_p:nNn {#4} = {#2} }
11276         { \int_compare_p:nNn {#2} = { -1 } }
11277         {#1}
11278         {#3}
11279       }

```



```
11280 }
```

(End of definition for `\file_full_name:n` and others. This function is documented on page 105.)

`\file_forget:n` Just a wrapper around a csname: we have to do a lookup here to make sure any paths are handled.

```
11281 \cs_new_protected:Npn \file_forget:n #1
11282 { \cs_undefine:c { __file_seen_ \file_full_name:n {#1} : } }
```

(End of definition for `\file_forget:n`. This function is documented on page 103.)

`\file_get_full_name:nN` These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers around the code above.

```
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF
  \_file_get_full_name_search:nN
11283 \cs_new_protected:Npn \file_get_full_name:nN #1#2
11284 {
11285   \file_get_full_name:nNF {#1} #2
11286   { \tl_set:Nn #2 { \q_no_value } }
11287 }
11288 \cs_generate_variant:Nn \file_get_full_name:nN { V }
11289 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11290 {
11291   \__kernel_tl_set:Nx #2
11292   { \file_full_name:n {#1} }
11293   \tl_if_empty:NTF #2
11294   { \prg_return_false: }
11295   { \prg_return_true: }
11296 }
11297 \prg_generate_conditional_variant:Nnn \file_get_full_name:nN
11298 { V } { T , F , TF }
```

(End of definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `\_file_get_full_name_search:nN`. These functions are documented on page 105.)

`\g__file_tmp_ior` A reserved stream to test for opening a shell.

```
11299 \ior_new:N \g__file_tmp_ior
```

(End of definition for `\g__file_tmp_ior`.)

`\file_md5five_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```
\file_md5five_hash:V
  \file_size:n
  \file_size:V
  \file_timestamp:n
  \file_timestamp:V
  \_file_details:nn
  \_file_details_aux:nn
  \_file_md5five_hash:n
11300 \cs_new:Npn \file_size:n #1
11301 { \_file_details:nn {#1} { size } }
11302 \cs_generate_variant:Nn \file_size:n { V }
11303 \cs_new:Npn \file_timestamp:n #1
11304 { \_file_details:nn {#1} { moddate } }
11305 \cs_generate_variant:Nn \file_timestamp:n { V }
11306 \cs_new:Npn \_file_details:nn #1#2
11307 {
11308   \exp_args:Ne \_file_details_aux:nn
11309   { \file_full_name:n {#1} } {#2}
11310 }
11311 \cs_new:Npn \_file_details_aux:nn #1#2
11312 {
```

```

11313 \tl_if_blank:nF {#1}
11314 { \use:c { tex_file #2 :D } {#1} }
11315 }
11316 \cs_new:Npn \file_md5five_hash:n #1
11317 { \exp_args:Ne \__file_md5five_hash:n { \file_full_name:n {#1} } }
11318 \cs_generate_variant:Nn \file_md5five_hash:n { V }
11319 \cs_new:Npn \__file_md5five_hash:n #1
11320 { \tex_md5fivesum:D file {#1} }

```

(End of definition for `\file_md5five_hash:n` and others. These functions are documented on page 104.)

`\file_hex_dump:nnn` These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```

\file_hex_dump:Vnn
\__file_hex_dump_auxi:nnn
\__file_hex_dump_auxii:nnnn
\__file_hex_dump_auxiii:nnnn
\__file_hex_dump_auxiiv:nnn
\file_hex_dump:n
\file_hex_dump:V
\__file_hex_dump:n
11321 \cs_new:Npn \file_hex_dump:nnn #1#2#3
11322 {
11323   \exp_args:Neee \__file_hex_dump_auxi:nnn
11324   { \file_full_name:n {#1} }
11325   { \int_eval:n {#2} }
11326   { \int_eval:n {#3} }
11327 }
11328 \cs_generate_variant:Nn \file_hex_dump:nnn { V }
11329 \cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
11330 {
11331   \bool_lazy_any:nF
11332   {
11333     { \tl_if_blank_p:n {#1} }
11334     { \int_compare_p:nNn {#2} = 0 }
11335     { \int_compare_p:nNn {#3} = 0 }
11336   }
11337   {
11338     \exp_args:Ne \__file_hex_dump_auxii:nnnn
11339     { \__file_details_aux:nn {#1} { size } }
11340     {#1} {#2} {#3}
11341   }
11342 }
11343 \cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
11344 {
11345   \int_compare:nNnTF {#3} > 0
11346   { \__file_hex_dump_auxiii:nnnn {#3} }
11347   {
11348     \exp_args:Ne \__file_hex_dump_auxiii:nnnn
11349     { \int_eval:n { #1 + #3 } }
11350   }
11351   {#1} {#2} {#4}
11352 }
11353 \cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
11354 {
11355   \int_compare:nNnTF {#4} > 0
11356   { \__file_hex_dump_auxiv:nnn {#4} }
11357   {
11358     \exp_args:Ne \__file_hex_dump_auxiv:nnn
11359     { \int_eval:n { #2 + #4 } }
11360   }
11361   {#1} {#3}

```

```

11362 }
11363 \cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
11364 {
11365   \tex_filedump:D
11366   offset ~ \int_eval:n { #2 - 1 } ~
11367   length ~ \int_eval:n { #1 - #2 + 1 }
11368   {#3}
11369 }
11370 \cs_new:Npn \file_hex_dump:n #1
11371 { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
11372 \cs_generate_variant:Nn \file_hex_dump:n { V }
11373 \sys_if_engine luatex:TF
11374 {
11375   \cs_new:Npn \__file_hex_dump:n #1
11376   {
11377     \tl_if_blank:nF {#1}
11378     { \tex_filedump:D whole {#1} {#1} }
11379   }
11380 }
11381 {
11382   \cs_new:Npn \__file_hex_dump:n #1
11383   {
11384     \tl_if_blank:nF {#1}
11385     { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
11386   }
11387 }

```

(End of definition for `\file_hex_dump:nnn` and others. These functions are documented on page 103.)

<pre> \file_get_hex_dump:nN \file_get_hex_dump:VN \file_get_hex_dump:nNTF \file_get_hex_dump:VNTF \file_get_mdfive_hash:nN \file_get_mdfive_hash:VN \file_get_mdfive_hash:nNTF \file_get_mdfive_hash:VNTF \file_get_size:nN \file_get_size:VN \file_get_size:nNTF \file_get_size:VNTF \file_get_timestamp:nN \file_get_timestamp:VN \file_get_timestamp:nNTF \file_get_timestamp:VNTF __file_get_details:nnN </pre>	Non-expandable wrappers around the above in the case where appropriate primitive support exists. <pre> 11388 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2 11389 { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } } 11390 \cs_generate_variant:Nn \file_get_hex_dump:nN { V } 11391 \cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2 11392 { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } } 11393 \cs_generate_variant:Nn \file_get_mdfive_hash:nN { V } 11394 \cs_new_protected:Npn \file_get_size:nN #1#2 11395 { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } } 11396 \cs_generate_variant:Nn \file_get_size:nN { V } 11397 \cs_new_protected:Npn \file_get_timestamp:nN #1#2 11398 { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } } 11399 \cs_generate_variant:Nn \file_get_timestamp:nN { V } 11400 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF } 11401 { __file_get_details:nnN {#1} { hex_dump } #2 } 11402 \prg_generate_conditional_variant:Nnn \file_get_hex_dump:nN 11403 { V } { T , F , TF } 11404 \prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF } 11405 { __file_get_details:nnN {#1} { mdfive_hash } #2 } 11406 \prg_generate_conditional_variant:Nnn \file_get_mdfive_hash:nN 11407 { V } { T , F , TF } 11408 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF } 11409 { __file_get_details:nnN {#1} { size } #2 } 11410 \prg_generate_conditional_variant:Nnn \file_get_size:nN </pre>

```

11411 { V } { T , F , TF }
11412 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
11413 { \_file_get_details:nnN {#1} { timestamp } #2 }
11414 \prg_generate_conditional_variant:Nnn \file_get_timestamp:nN
11415 { V } { T , F , TF }
11416 \cs_new_protected:Npn \_file_get_details:nnN #1#2#3
11417 {
11418   \__kernel_tl_set:Nx #3
11419   { \use:c { file_ #2 :n } {#1} }
11420   \tl_if_empty:NTF #3
11421   { \prg_return_false: }
11422   { \prg_return_true: }
11423 }

```

(End of definition for \file_get_hex_dump:nNTF and others. These functions are documented on page 104.)

\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF

Custom code due to the additional arguments.

```

11424 \cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
11425 {
11426   \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
11427   { \tl_set:Nn #4 { \q_no_value } }
11428 }
11429 \cs_generate_variant:Nn \file_get_hex_dump:nnnN { V }
11430 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
11431 { T , F , TF }
11432 {
11433   \__kernel_tl_set:Nx #4
11434   { \file_hex_dump:nnn {#1} {#2} {#3} }
11435   \tl_if_empty:NTF #4
11436   { \prg_return_false: }
11437   { \prg_return_true: }
11438 }
11439 \prg_generate_conditional_variant:Nnn \file_get_hex_dump:nnnN
11440 { V } { T , F , TF }

```

(End of definition for \file_get_hex_dump:nnnNTF. This function is documented on page 104.)

_file_str_cmp:nn

As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```

11441 \cs_new_eq:NN \_file_str_cmp:nn \tex_strcmp:D

```

(End of definition for _file_str_cmp:nn.)

\file_compare_timestamp:p:nNn

Comparison of file date can be done by using the low-level nature of the string comparison functions.

\file_compare_timestamp:p:nNV

\file_compare_timestamp:p:VNn

\file_compare_timestamp:p:VNV

\file_compare_timestamp:nNnTF

\file_compare_timestamp:nNVTF

\file_compare_timestamp:VNnTF

\file_compare_timestamp:VNVTF

_file_compare_timestamp:nnN

_file_timestamp:n

```

11442 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11443 { p , T , F , TF }
11444 {
11445   \exp_args:Nee \_file_compare_timestamp:nnN
11446   { \file_full_name:n {#1} }
11447   { \file_full_name:n {#3} }
11448   #2
11449 }
11450 \prg_generate_conditional_variant:Nnn \file_compare_timestamp:nNn

```

```

11451 { nNV , V , VNV } { p , T , F , TF }
11452 \cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
11453 {
11454   \tl_if_blank:nTF {#1}
11455   {
11456     \if_charcode:w #3 <
11457     \prg_return_true:
11458   \else:
11459     \prg_return_false:
11460   \fi:
11461 }
11462 {
11463   \tl_if_blank:nTF {#2}
11464   {
11465     \if_charcode:w #3 >
11466     \prg_return_true:
11467   \else:
11468     \prg_return_false:
11469   \fi:
11470 }
11471 {
11472   \if_int_compare:w
11473   \__file_str_cmp:nn
11474   { \__file_timestamp:n {#1} }
11475   { \__file_timestamp:n {#2} }
11476   #3 \c_zero_int
11477   \prg_return_true:
11478 \else:
11479   \prg_return_false:
11480 \fi:
11481 }
11482 }
11483 }
11484 \cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D

```

(End of definition for \file_compare_timestamp:nNnTF, __file_compare_timestamp:nnN, and __file_timestamp:n. This function is documented on page 105.)

<code>\file_if_exist_p:n</code> <code>\file_if_exist_p:V</code> <code>\file_if_exist:nTF</code> <code>\file_if_exist:VTF</code>	The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.	<pre> 11485 \prg_new_conditional:Npnn \file_if_exist:n #1 { p , T , F , TF } 11486 { 11487 \tl_if_blank:eTF { \file_full_name:n {#1} } 11488 { \prg_return_false: } 11489 { \prg_return_true: } 11490 } 11491 \prg_generate_conditional_variant:Nnn \file_if_exist:n { V } { p , T , F , TF } </pre>

(End of definition for \file_if_exist:nTF. This function is documented on page 103.)

<code>\file_if_exist_input:n</code> <code>\file_if_exist_input:V</code> <code>\file_if_exist_input:nF</code> <code>\file_if_exist_input:VF</code>	Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the <code><true code></code> would be inconsistent with other conditionals.	<pre> 11492 \cs_new_protected:Npn \file_if_exist_input:n #1 </pre>

```

11493 {
11494   \file_get_full_name:nNT {#1} \l__file_full_name_tl
11495   { \__file_input:V \l__file_full_name_tl }
11496 }
11497 \cs_generate_variant:Nn \file_if_exist_input:n { V }
11498 \cs_new_protected:Npn \file_if_exist_input:nF #1#2
11499 {
11500   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11501   { \__file_input:V \l__file_full_name_tl }
11502   {#2}
11503 }
11504 \cs_generate_variant:Nn \file_if_exist_input:nF { V }

```

(End of definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 106.)

`\file_input_stop:` A simple rename.

```

11505 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }

```

(End of definition for `\file_input_stop:`. This function is documented on page 107.)

`\__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```

11506 \cs_new_protected:Npn \__kernel_file_missing:n #1
11507 {
11508   \msg_error:nne { kernel } { file-not-found }
11509   { \__kernel_file_name_sanitiz:n {#1} }
11510 }

```

(End of definition for `\__kernel_file_missing:n`.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

`\file_input:V`

`\__file_input:n`

`\__file_input:V`

`\__file_input_push:n`

`\__kernel_file_input_push:n`

`\__file_input_pop:`

`\__kernel_file_input_pop:`

`\__file_input_pop:nnn`

```

11511 \cs_new_protected:Npn \file_input:n #1
11512 {
11513   \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11514   { \__file_input:V \l__file_full_name_tl }
11515   { \__kernel_file_missing:n {#1} }
11516 }
11517 \cs_generate_variant:Nn \file_input:n { V }
11518 \cs_new_protected:Npe \__file_input:n #1
11519 {
11520   \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
11521   { \exp_not:N \@addtofilelist {#1} }
11522   { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
11523   \exp_not:N \__file_input_push:n {#1}
11524   \exp_not:N \tex_input:D
11525   \sys_if_engine luatex:TF
11526   { {#1} }
11527   { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11528   \exp_not:N \__file_input_pop:
11529 }
11530 \cs_generate_variant:Nn \__file_input:n { V }

```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

11531 \cs_new_protected:Npn \__file_input_push:n #1
11532 {
11533   \seq_gpush:Ne \g__file_stack_seq
11534   {
11535     { \g_file_curr_dir_str }
11536     { \g_file_curr_name_str }
11537     { \g_file_curr_ext_str }
11538   }
11539   \file_parse_full_name:nNNN {#1}
11540   \l__file_dir_str \l__file_name_str \l__file_ext_str
11541   \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
11542   \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
11543   \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
11544 }
11545 \cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
11546 \cs_new_protected:Npn \__file_input_pop:
11547 {
11548   \seq_gpop:NN \g__file_stack_seq \l__file_tmp_tl
11549   \exp_after:wN \__file_input_pop:nnn \l__file_tmp_tl
11550 }
11551 \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
11552 \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
11553 {
11554   \str_gset:Nn \g_file_curr_dir_str {#1}
11555   \str_gset:Nn \g_file_curr_name_str {#2}
11556   \str_gset:Nn \g_file_curr_ext_str {#3}
11557 }

```

(End of definition for \file_input:n and others. This function is documented on page 106.)

```

\file_input_raw:n No error checking, no tracking.
\file_input_raw:V
\__file_input_raw:nn
11558 \cs_new:Npn \file_input_raw:n #1
11559 { \exp_args:Ne \__file_input_raw:nn { \file_full_name:n {#1} } {#1} }
11560 \cs_generate_variant:Nn \file_input_raw:n { V }
11561 \cs_new:Npn \__file_input_raw:nn #1#2
11562 {
11563   \tl_if_blank:nTF {#1}
11564   {
11565     \exp_args:Nnne \msg_expandable_error:nnn
11566     { kernel } { file-not-found }
11567     { \__kernel_file_name_sanitiz:n {#2} }
11568   }
11569   { \tex_input:D {#1} }
11570 }
11571 \exp_args_generate:n { nne }

```

(End of definition for \file_input_raw:n and __file_input_raw:nn. This function is documented on page 106.)

```

\file_parse_full_name:n The main parsing macro \file_parse_full_name_apply:nN passes the file name #1
\file_parse_full_name:V through \__kernel_file_name_sanitiz:n so that we have a single normalized way
\file_parse_full_name_apply:nN
\file_parse_full_name_apply:VN

```

to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```

11572 \cs_new:Npn \file_parse_full_name:n #1
11573 {
11574     \file_parse_full_name_apply:nN {#1}
11575     \prg_do_nothing:
11576 }
11577 \cs_generate_variant:Nn \file_parse_full_name:n { V }
11578 \cs_new:Npn \file_parse_full_name_apply:nN #1
11579 {
11580     \exp_args:Ne \__file_parse_full_name_auxi:nN
11581     { \__kernel_file_name_sanitiz:n {#1} }
11582 }
11583 \cs_generate_variant:Nn \file_parse_full_name_apply:nN { V }

```

`\__file_parse_full_name_area:nw` splits the file name into chunks separated by `/`, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When `\__file_parse_full_name_area:nw` is done, it leaves the path within braces after the scan mark `\s__file_stop` and proceeds parsing the actual file name.

```

\__file_parse_full_name_auxi:nN
\__file_parse_full_name_area:nw
11584 \cs_new:Npn \__file_parse_full_name_auxi:nN #1
11585 {
11586     \__file_parse_full_name_area:nw { } #1
11587     / \s__file_stop
11588 }
11589 \cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
11590 {
11591     \tl_if_empty:nTF {#3}
11592     { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
11593     { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
11594 }

```

`\__file_parse_full_name_base:nw` does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case #1 is empty, it is assumed that the extension is actually empty, and the file name is #2. Besides, an extra `.` has to be added to #2 because it is later removed in `\__file_parse_full_name_tidy:nnnN`. In any case, if there's an extension, it is returned with a leading `..`

```

\__file_parse_full_name_base:nw
11595 \cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
11596 {
11597     \tl_if_empty:nTF {#3}
11598     {
11599         \tl_if_empty:nTF {#1}
11600         {
11601             \tl_if_empty:nTF {#2}
11602             { \__file_parse_full_name_tidy:nnnN { } { } }
11603             { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
11604         }
11605         { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
11606     }
11607     { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
11608 }

```


Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading / and . in the file path an name, so here we need to remove them, except in the path, if it is a single /, in which case it's left as is. After all's done, pass to #4.

```

11609 \cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
11610 {
11611   \exp_args:Nee #4
11612   {
11613     \str_if_eq:nnF {#3} { / } { \use_none:n }
11614     #3 \prg_do_nothing:
11615   }
11616   { \use_none:n #1 \prg_do_nothing: }
11617   {#2}
11618 }

```

(End of definition for \file_parse_full_name:n and others. These functions are documented on page 106.)

\file_parse_full_name:nNNN

\file_parse_full_name:VNNN

```

11619 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
11620 {
11621   \file_parse_full_name_apply:nN {#1}
11622   \__file_full_name_assign:nnnNNN #2 #3 #4
11623 }
11624 \cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
11625 {
11626   \str_set:Nn #4 {#1}
11627   \str_set:Nn #5 {#2}
11628   \str_set:Nn #6 {#3}
11629 }
11630 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }

```

(End of definition for \file_parse_full_name:nNNN. This function is documented on page 105.)

\file_show_list:

\file_log_list:

__file_list:N

__file_list_aux:n

A function to list all files used to the log, without duplicates. In package mode, if \@filelist is still defined, we need to take this list of file names into account (we capture it \@AtBeginDocument into \g__file_record_seq), turning it to a string (this does not affect the commas of this comma list).

```

11631 \cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nneeee }
11632 \cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nneeee }
11633 \cs_new_protected:Npn \__file_list:N #1
11634 {
11635   \seq_clear:N \l__file_tmp_seq
11636   \clist_if_exist:NT \@filelist
11637   {
11638     \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
11639     { \tl_to_str:N \@filelist }
11640   }
11641   \seq_concat:Nnn \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
11642   \seq_remove_duplicates:N \l__file_tmp_seq
11643   #1 { kernel } { file-list }
11644   { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
11645   { } { } { }
11646 }
11647 \cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }

```

(End of definition for `\file_show_list:` and others. These functions are documented on page 107.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```

11648 \cs_if_exist:NT \@filelist
11649 {
11650   \AtBeginDocument
11651   {
11652     \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
11653     { \tl_to_str:N \@filelist }
11654     \seq_gconcat:NNN
11655       \g__file_record_seq
11656       \g__file_record_seq
11657       \l__file_tmp_seq
11658   }
11659 }
```

54.5 GetIdInfo

`\GetIdInfo` As documented in `expl3.dtx` this function extracts file name etc from an SVN Id line. This used to be how we got version number and so on in all modules, so it had to be defined in `l3bootstrap`. Now it's more convenient to define it after we have set up quite a lot of tools, and `l3file` seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the Id keyword!

```

11660 \cs_new_protected:Npn \GetIdInfo
11661 {
11662   \tl_clear_new:N \ExplFileDescription
11663   \tl_clear_new:N \ExplFileDate
11664   \tl_clear_new:N \ExplFileName
11665   \tl_clear_new:N \ExplFileExtension
11666   \tl_clear_new:N \ExplFileVersion
11667   \group_begin:
11668   \char_set_catcode_space:n { 32 }
11669   \exp_after:wN
11670   \group_end:
11671   \__file_id_info_auxi:w
11672 }
```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `\__file_id_info_auxii:w`.

```

11673 \cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
11674 {
11675   \tl_set:Nn \ExplFileDescription {#2}
11676   \str_if_eq:nnTF {#1} { Id }
11677   {
11678     \tl_set:Nn \ExplFileDate { 0000/00/00 }
11679     \tl_set:Nn \ExplFileName { [unknown] }
11680     \tl_set:Nn \ExplFileExtension { [unknown~extension] }

```

```

11681         \tl_set:Nn \ExplFileVersion {-1}
11682     }
11683     { \__file_id_info_auxii:w #1 ~ \s__file_stop }
11684 }

```

Here, #1 is Id, #2 is the file name, #3 is the extension, #4 is the version, #5 is the check in date and #6 is the check in time and user, plus some trailing spaces. If #4 is the marker -1 value then #5 and #6 are empty.

```

11685 \cs_new_protected:Npn \__file_id_info_auxii:w
11686     #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
11687 {
11688     \tl_set:Nn \ExplFileName {#2}
11689     \tl_set:Nn \ExplFileExtension {#3}
11690     \tl_set:Nn \ExplFileVersion {#4}
11691     \str_if_eq:nnTF {#4} {-1}
11692     { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
11693     { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
11694 }

```

Convert an SVN-style date into a L^AT_EX-style one.

```

11695 \cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
11696 { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }

```

(End of definition for \GetIdInfo and others. This function is documented on page 11.)

54.6 Checking the version of kernel dependencies

```

\__kernel_dependency_version_check:Nn
\__kernel_dependency_version_check:nn
\__file_kernel_dependency_compare:nnn
\__file_parse_version:w

```

This function is responsible for checking if dependencies of the L^AT_EX3 kernel match the version preloaded in the L^AT_EX2_ε kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```

11697 \cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
11698 { \exp_args:NV \__kernel_dependency_version_check:nn #1 }
11699 \cs_new_protected:Npn \__kernel_dependency_version_check:nn #1
11700 {
11701     \cs_if_exist:NTF \c__kernel_expl_date_tl
11702     {
11703         \exp_args:NV \__file_kernel_dependency_compare:nnn
11704         \c__kernel_expl_date_tl {#1}
11705     }
11706     { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
11707 }
11708 \cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
11709 {
11710     \int_compare:nNtT
11711     { \__file_parse_version:w #1 \s__file_stop } <
11712     { \__file_parse_version:w #2 \s__file_stop }
11713     { \__file_mismatched_dependency_error:nn {#2} {#3} }
11714 }
11715 \cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}

```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich -all -engine=<engine> <format>[-dev].fmt` to query the format files available. A shell is opened and each line is read into a sequence.

```

\__file_mismatched_dependency_error:nn
11716 \cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
11717 {
11718   \exp_args:NNe \ior_shell_open:Nn \g__file_tmp_ior
11719   {
11720     kpsewhich ~ --all ~
11721     --engine = \c_sys_engine_exec_str
11722     \c_space_tl \c_sys_engine_format_str
11723     \bool_lazy_and:nnT
11724       { \tl_if_exist_p:N \development@branch@name }
11725       { ! \tl_if_empty_p:N \development@branch@name }
11726     { -dev } .fmt
11727   }
11728   \seq_clear:N \l__file_tmp_seq
11729   \ior_map_inline:Nn \g__file_tmp_ior
11730     { \seq_put_right:Nn \l__file_tmp_seq {##1} }
11731   \ior_close:N \g__file_tmp_ior
11732   \msg_error:nnnn { kernel } { mismatched-support-file }
11733   {#1} {#2}

```

And finish by ending the current file.

```

11734   \tex_endinput:D
11735 }

```

Now define the actual error message:

```

11736 \msg_new:nnnn { kernel } { mismatched-support-file }
11737 {
11738   Mismatched-LaTeX-support-files-detected. \\
11739   Loading~'#2'~aborted!

```

`\c__kernel_expl_date_tl` may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```

11740   \tl_if_exist:NT \c__kernel_expl_date_tl
11741   {
11742     \\ \\
11743     The~L3~programming~layer~in~the~LaTeX~format \\
11744     is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~
11745     tree~the~files~require \\ at~least~#1.
11746   }
11747 }
11748 {

```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```

11749   \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1
11750   {
11751     The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
11752     LaTeX~found~these~files:
11753     \seq_map_tokens:Nn \l__file_tmp_seq { \\---\use:n } \\

```

```

11754     Try-deleting-the~file~in~the~user~tree~then~run~LaTeX~again.
11755   }
11756   {
11757     The~most~likely~causes~are:
11758     \\~~~A~recent~format~generation~failed;
11759     \\~~~A~stray~format~file~in~the~user~tree~which~needs~
11760       to~be~removed~or~rebuilt;
11761     \\~~~You~are~running~a~manually~installed~version~of~#2 \\
11762     \ \ \ which~is~incompatible~with~the~version~in~LaTeX. \\
11763   }
11764   \\
11765   LaTeX~will~abort~loading~the~incompatible~support~files~
11766   but~this~may~lead~to \\ later~errors.~Please~ensure~that~
11767   your~LaTeX~format~is~correctly~regenerated.
11768 }

```

(End of definition for `\__kernel_dependency_version_check:Nn` and others.)

54.7 Messages

```

11769 \msg_new:nnnn { kernel } { file-not-found }
11770 { File~'#1'~not~found. }
11771 {
11772   The~requested~file~could~not~be~found~in~the~current~directory,~
11773   in~the~TeX~search~path~or~in~the~LaTeX~search~path.
11774 }
11775 \msg_new:nnn { kernel } { file-list }
11776 {
11777   >~File~List~<
11778   #1 \\
11779   .....
11780 }
11781 \msg_new:nnnn { kernel } { filename-chars-lost }
11782 { #1~invalid~in~file~name.~Lost:~#2. }
11783 {
11784   There~was~an~invalid~token~in~the~file~name~that~caused~
11785   the~characters~following~it~to~be~lost.
11786 }
11787 \msg_new:nnnn { kernel } { filename-missing-endcsname }
11788 { Missing~\iow_char:N\\endcsname~inserted~in~filename. }
11789 {
11790   The~file~name~had~more~\iow_char:N\\csname~commands~than~
11791   \iow_char:N\\endcsname~ones.~LaTeX~will~add~the~missing~
11792   \iow_char:N\\endcsname~and~try~to~continue~as~best~as~it~can.
11793 }
11794 \msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
11795 { Unbalanced~quotes~in~file~name~'#1'. }
11796 {
11797   File~names~must~contain~balanced~numbers~of~quotes~(").
11798 }
11799 \msg_new:nnnn { kernel } { iow-indent }
11800 { Only~#1~allows~#2 }
11801 {
11802   The~command~#2~can~only~be~used~in~messages~

```

```

11803     which~will~be~wrapped~using~#1.
11804     \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
11805 }

```

54.8 Functions delayed from earlier modules

<@@=sys>

\c_sys_platform_str Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `ifplatform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```

11806 \sys_if_engine luatex:TF
11807 {
11808     \str_const:Ne \c_sys_platform_str
11809     { \tex_directlua:D { tex.print(os.type) } }
11810 }
11811 {
11812     \file_if_exist:nTF { nul: }
11813     {
11814         \file_if_exist:nF { /dev/null }
11815         { \str_const:Nn \c_sys_platform_str { windows } }
11816     }
11817     {
11818         \file_if_exist:nT { /dev/null }
11819         { \str_const:Nn \c_sys_platform_str { unix } }
11820     }
11821 }
11822 \cs_if_exist:NF \c_sys_platform_str
11823 { \str_const:Nn \c_sys_platform_str { unknown } }

```

(End of definition for `\c_sys_platform_str`. This variable is documented on page 79.)

\sys_if_platform_unix_p: We can now set up the tests.

```

11824 \clist_map_inline:nn { unix , windows }
11825 {
11826     \__file_const:nn { sys_if_platform_ #1 }
11827     { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
11828 }

```

(End of definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 79.)

11829 </code>

Chapter 55

l3luatex implementation

```
11830 <@@=lua>
```

55.1 Breaking out to Lua

```
11831 <*code>
```

```
\__lua_escape:n Copies of primitives.
\__lua_now:n      11832 \cs_new_eq:NN \__lua_escape:n \tex_luaescapestring:D
\__lua_shipout:n 11833 \cs_new_eq:NN \__lua_now:n      \tex_directlua:D
                  11834 \cs_new_eq:NN \__lua_shipout:n \tex_latelua:D
```

(End of definition for __lua_escape:n, __lua_now:n, and __lua_shipout:n.)

These functions are set up in l3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

```
11835 \cs_undefine:N \lua_escape:e
11836 \cs_undefine:N \lua_now:e
```

```
\lua_now:n Wrappers around the primitives.
\lua_now:e  11837 \cs_new:Npn \lua_now:e #1 { \__lua_now:n {#1} }
\lua_shipout_e:n 11838 \cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }
\lua_shipout:n  11839 \cs_new_protected:Npn \lua_shipout_e:n #1 { \__lua_shipout:n {#1} }
\lua_escape:n    11840 \cs_new_protected:Npn \lua_shipout:n #1
\lua_escape:e    11841 { \lua_shipout_e:n { \exp_not:n {#1} } }
                  11842 \cs_new:Npn \lua_escape:e #1 { \__lua_escape:n {#1} }
                  11843 \cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }
```

(End of definition for \lua_now:n and others. These functions are documented on page 108.)

```
\lua_load_module:n Wrapper around require'<module>'.
                  11844 \str_new:N \l__lua_err_msg_str
                  11845 \cs_new_protected:Npn \lua_load_module:n #1
                  11846 {
                  11847   \bool_if:nF { \__lua_load_module_p:n { #1 } }
                  11848   {
                  11849     \msg_error:nnnV
                  11850     { luatex } { module-not-found } { #1 } \l__lua_err_msg_str
                  11851   }
                  11852 }
```

(End of definition for `\lua_load_module:n`. This function is documented on page 109.)

As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/

```

11853 \sys_if_engine luatex:F
11854 {
11855   \clist_map_inline:nn
11856   {
11857     \lua_escape:n , \lua_escape:e ,
11858     \lua_now:n , \lua_now:e
11859   }
11860   {
11861     \cs_gset:Npn #1 ##1
11862     {
11863       \msg_expandable_error:nnn
11864       { luatex } { luatex-required } { #1 }
11865     }
11866   }
11867   \clist_map_inline:nn
11868   { \lua_shipout_e:n , \lua_shipout:n, \lua_load_module:n }
11869   {
11870     \cs_gset_protected:Npn #1 ##1
11871     {
11872       \msg_error:nnn
11873       { luatex } { luatex-required } { #1 }
11874     }
11875   }
11876 }

```

55.2 Messages

```

11877 \msg_new:nnnn { luatex } { luatex-required }
11878 { LuaTeX-engine-not-in-use!~Ignoring~#1. }
11879 {
11880   The~feature~you~are~using~is~only~available~
11881   with~the~LuaTeX-engine.~LaTeX3~ignored~'~#1~'.
11882 }
11883 \msg_new:nnnn { luatex } { module-not-found }
11884 { Lua-module~'~#1~'~not~found. }
11885 {
11886   The~file~'~#1.lua~'~could~not~be~found.~Please~ensure~
11887   that~the~file~was~properly~installed~and~that~the~
11888   filename~database~is~current. \\ \\
11889   The~Lua~loader~provided~this~additional~information: \\
11890   #2
11891 }
11892 \prop_gput:Nnn \g_msg_module_name_prop { luatex } { LaTeX }
11893 \prop_gput:Nnn \g_msg_module_type_prop { luatex } { }
11894 \code

```

55.3 Lua functions for internal use

```

11895 (*lua)

```


Most of the emulation of pdfTeX here is based heavily on Heiko Oberdiek's pdfTeX-cmds package.

`ltx.utils` Create a table for the kernel's own use.

```
11896 ltx = ltx or {utils={}}
11897 ltx.utils = ltx.utils or { }
11898 local ltxutils = ltx.utils
```

(End of definition for `ltx.utils`. This function is documented on page 109.)

Local copies of global tables.

```
11899 local io      = io
11900 local kpse    = kpse
11901 local lfs     = lfs
11902 local math    = math
11903 local md5     = md5
11904 local os      = os
11905 local string  = string
11906 local tex     = tex
11907 local texio   = texio
11908 local tonumber = tonumber
11909 local token   = token
```

Local copies of standard functions.

```
11910 local abs      = math.abs
11911 local byte     = string.byte
11912 local floor    = math.floor
11913 local format   = string.format
11914 local gsub     = string.gsub
11915 local lfs_attr = lfs.attributes
11916 local open     = io.open
11917 local os_date  = os.date
11918 local setcatcode = tex.setcatcode
11919 local sprint   = tex.sprint
11920 local cprint   = tex.cprint
11921 local write    = tex.write
11922 local write_nl = texio.write_nl
11923 local utf8_char = utf8.char
11924 local package_loaded = package.loaded
11925 local package_searchers = package.searchers
11926 local table_concat = table.concat
11927
11928 local scan_csname = token.scancsname or token.scan_csname
11929 local scan_int    = token.scan_int or token.scan_integer
11930 local scan_string = token.scanstring or token.scan_string
11931 local scan_keyword = token.scankeyword or token.scan_keyword
11932 local scan_argument = token.scanargument or token.scan_argument
11933 local get_next    = token.scannext or token.get_next
11934 local put_next    = token.putnext or token.put_next
11935 local token_create = token.create
11936 local get_macro   = token.getmacro or token.get_macro
11937 local get_protected = token.getprotected or token.get_protected
11938 local get_csname  = token.getcsname or token.get_csname
11939 local token_new   = token.new
11940 local set_macro   = token.setmacro or token.set_macro
11941
```

```

11942 local active_prefix = status.luatex_engine == 'luametateX' and status.getconstants().active
11943
11944 local lbrace, rbrace = token_create(byte'{'}, token_create(byte'}')

```

Since token.create only returns useful values after the tokens has been added to TeX's hash table, we define a variant which defines it first if necessary.

```

11945 local token_create_safe
11946 do
11947   local is_defined = token.is_defined
11948   local set_char   = token.set_char or tex.chardef
11949   local runtoks    = tex.runtoks
11950   local let_token  = token_create'let'
11951
11952   function token_create_safe(s)
11953     local orig_token = token_create(s)
11954     if is_defined(s, true) then
11955       return orig_token
11956     end
11957     set_char(s, 0)
11958     local new_token = token_create(s)
11959     runtoks(function()
11960       put_next(let_token, new_token, orig_token)
11961     end)
11962     return new_token
11963   end
11964 end
11965
11966 local true_tok    = token_create_safe'prg_return_true:'
11967 local false_tok   = token_create_safe'prg_return_false:'

```

In ConTeXt lmtx token.command_id does not exist, but it can easily be emulated with ConTeXt's tokens.commands.

```

11968 local command_id  = token.command_id
11969 if not command_id and tokens and tokens.commands then
11970   local id_map = tokens.commands
11971   function command_id(name)
11972     return id_map[name]
11973   end
11974 end

```

Deal with ConTeXt: doesn't use kpse library.

```

11975 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file

```

escapehex An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdftexcmds but is not currently required here.

```

11976 local function escapehex(str)
11977   return (gsub(str, ".",
11978     function (ch) return format("%02X", byte(ch)) end))
11979 end

```

(End of definition for escapehex.)

ltx.utils.filedump Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```

11980 local function filedump(name,offset,length)
11981   local file = kpse_find(name,"tex",true)
11982   if not file then return end
11983   local f = open(file,"rb")
11984   if not f then return end
11985   if offset and offset > 0 then
11986     f:seek("set", offset)
11987   end
11988   local data = f:read(length or 'a')
11989   f:close()
11990   return escapehex(data)
11991 end
11992 ltxutils.filedump = filedump

```

(End of definition for ltx.utils.filedump. This function is documented on page 109.)

md5.HEX Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is built-in. For traditional Lua_T_E_X, the conversion to hexadecimal has to be done by us.

```

11993 local md5_HEX = md5.HEX
11994 if not md5_HEX then
11995   local md5_sum = md5.sum
11996   function md5_HEX(data)
11997     return escapehex(md5_sum(data))
11998   end
11999   md5.HEX = md5_HEX
12000 end

```

(End of definition for md5.HEX.)

ltx.utils.filemd5sum Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdftexcmds` and how it handles strings that have passed through Lua_T_E_X). The file is read in binary mode so that no line ending normalization occurs.

```

12001 local function filemd5sum(name)
12002   local file = kpse_find(name, "tex", true) if not file then return end
12003   local f = open(file, "rb") if not f then return end
12004
12005   local data = f:read("*a")
12006   f:close()
12007   return md5_HEX(data)
12008 end
12009 ltxutils.filemd5sum = filemd5sum

```

(End of definition for ltx.utils.filemd5sum. This function is documented on page 109.)

ltx.utils.filemoddate There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with Z.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of pdf_T_E_X.

```

12010 local filemoddate
12011 if os_date'%z':match'^[+-%d%d%d$d$' then
12012   local pattern = lpeg.Cs(16 *

```

```

12013      (lpeg.Cg(lpeg.S'+-' * '0000' * lpeg.Cc'Z')
12014      + 3 * lpeg.Cc'"'" * 2 * lpeg.Cc'"'"
12015      + lpeg.Cc'Z')
12016      * -1)
12017  function filemoddate(name)
12018      local file = kpse_find(name, "tex", true)
12019      if not file then return end
12020      local date = lfs_attr(file, "modification")
12021      if not date then return end
12022      return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))
12023  end
12024  else
12025      local function filemoddate(name)
12026          local file = kpse_find(name, "tex", true)
12027          if not file then return end
12028          local date = lfs_attr(file, "modification")
12029          if not date then return end
12030          local d = os_date("*t", date)
12031          local u = os_date("!*t", date)
12032          local off = 60 * (d.hour - u.hour) + d.min - u.min
12033          if d.year ~= u.year then
12034              if d.year > u.year then
12035                  off = off + 1440
12036              else
12037                  off = off - 1440
12038              end
12039          elseif d.yday ~= u.yday then
12040              if d.yday > u.yday then
12041                  off = off + 1440
12042              else
12043                  off = off - 1440
12044              end
12045          end
12046          local timezone
12047          if off == 0 then
12048              timezone = "Z"
12049          else
12050              if off < 0 then
12051                  timezone = "-"
12052                  off = -off
12053              else
12054                  timezone = "+"
12055              end
12056              timezone = format("%s%02d'%02d'", timezone, hours // 60, hours % 60)
12057          end
12058          return format("D:%04d%02d%02d%02d%02d%02d%s",
12059              d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
12060      end
12061  end
12062  ltxutils.filemoddate = filemoddate

```

(End of definition for `ltx.utils.filemoddate`. This function is documented on page [109](#).)

`ltx.utils.filesize` A simple disk lookup.

```

12063 local function filesize(name)
12064     local file = kpse_find(name, "tex", true)
12065     if file then
12066         local size = lfs_attr(file, "size")
12067         if size then
12068             return size
12069         end
12070     end
12071 end
12072 ltxutils.filesize = filesize

```

(End of definition for ltx.utils.filesize. This function is documented on page 110.)

`luacmd` An internal function for defining control sequences from Lua which behave like primitives. This acts as a wrapper around `token.set_lua` which accepts a function instead of an index into the functions table.

```

12073 local luacmd do
12074     local set_lua = token.setlua or token.set_lua
12075     local undefined_cs = command_id'undefined_cs'
12076
12077     if not context and not luatexbase then require'ltluatex' end
12078     if luatexbase then
12079         local new_luafunction = luatexbase.new_luafunction
12080         local functions = lua.get_functions_table()
12081         function luacmd(name, func, ...)
12082             local id
12083             local tok = token_create(name)
12084             if tok.command == undefined_cs then
12085                 id = new_luafunction(name)
12086                 set_lua(name, id, ...)
12087             else
12088                 id = tok.index or tok.mode
12089             end
12090             functions[id] = func
12091         end
12092     elseif context then
12093         local register = context.functions.register
12094         local functions = context.functions.known
12095         function luacmd(name, func, ...)
12096             local tok = token_create(name)
12097             if tok.command == undefined_cs then
12098                 set_lua(name, register(func), ...)
12099             else
12100                 functions[tok.index or tok.mode] = func
12101             end
12102         end
12103     end
12104 end

```

(End of definition for luacmd.)

`try_require` Loads a Lua module. This function loads the module similarly to the standard Lua global function `require`, with a few differences. On success, `try_require` returns

`true`, `module`. If the module cannot be found, it returns `false`, `err_msg`. If the module is found, but something goes wrong when loading it, the function throws an error.

```

12105 local function try_require(name)
12106   if package_loaded[name] then
12107     return true, package_loaded[name]
12108   end
12109
12110   local failure_details = {}
12111   for _, searcher in ipairs(package_searchers) do
12112     local loader, data = searcher(name)
12113     if type(loader) == 'function' then
12114       package_loaded[name] = loader(name, data) or true
12115       return true, package_loaded[name]
12116     elseif type(loader) == 'string' then
12117       failure_details[#failure_details + 1] = loader
12118     end
12119   end
12120
12121   return false, table_concat(failure_details, '\n')
12122 end

```

(End of definition for `try_require`.)

`\__lua_load_module_p:n` Check to see if we can load a module using `require`. If we can load the module, then we load it immediately. Otherwise, we save the error message in `\l_@@_err_msg_str`.

```

12123 local char_given = command_id'char_given'
12124 local c_true_bool = token_create(1, char_given)
12125 local c_false_bool = token_create(0, char_given)
12126 local c_str_cctab = token_create('c_str_cctab').mode
12127
12128 luacmd('__lua_load_module_p:n', function()
12129   local success, result = try_require(scan_string())
12130   if success then
12131     set_macro(c_str_cctab, 'l__lua_err_msg_str', '')
12132     put_next(c_true_bool)
12133   else
12134     set_macro(c_str_cctab, 'l__lua_err_msg_str', result)
12135     put_next(c_false_bool)
12136   end
12137 end)

```

(End of definition for `\__lua_load_module_p:n`.)

55.4 Preserving iniTeX Lua data for runs

The Lua state is not dumped when a format is written, therefore any Lua variables filled doing format building need to be restored in order to be accessible during normal runs.

We provide some kernel-internal helpers for this. They will only be available if `luatexbase` is available. This is not a big restriction though, because ConTeXt (which does not use `luatexbase`) does not load `expl3` in the format.

```

12138 local register_luadata, get_luadata
12139

```

```

12140 if luatexbase then
12141   local register = token_create'@expl@luadata@bytecode'.index

```

`register_luadata` `register_luadata` is only available during format generation. It accept a string which uniquely identifies the data object and has to be provided to retrieve it later. Additionally it accepts a function which is called in the `pre_dump` callback and which has to return a string that evaluates to a valid Lua object to be preserved. Note that format generation does not necessarily mean the first run, because some packages such as `mylatexformat` generates a format based on an existing format.

```

12142   if status.ini_version then
12143     local luadata, luadata_order = {}, {}
12144
12145     function register_luadata(name, func)
12146       if luadata[name] then
12147         error(format("LaTeX error: data name %q already in use", name))
12148       end
12149       luadata[name] = func
12150       luadata_order[#luadata_order + 1] = func and name
12151     end

```

(End of definition for register_luadata.)

The actual work is done in `pre_dump`. The `luadata_order` is used to ensure that the order is consistent over multiple runs.

```

12152   luatexbase.add_to_callback("pre_dump", function()
12153     if next(luadata) then
12154       local str = "return {"
12155       for i=1, #luadata_order do
12156         local name = luadata_order[i]
12157         str = format('%s[%q]=%s,', str, name, luadata[name]())
12158       end
12159       lua.bytecode[register] = assert(load(str .. "}"))
12160     end
12161     end, "ltx.luadata")
12162   end

```

`get_luadata` `get_luadata` is only available if data should be restored. It accept the identifier which was used when the data object was registered and returns the associated object. Every object can only be retrieved once. Note that it is possible for both `register_luadata` and `get_luadata` to be available, for example while compiling a precompiled preamble. In such a case, data must be registered again to be kept in the next run.

```

12163   local luadata = lua.bytecode[register]
12164   if luadata then
12165     lua.bytecode[register] = nil
12166     luadata = luadata()
12167     function get_luadata(name)
12168       if not luadata then return end
12169       local data = luadata[name]
12170       luadata[name] = nil
12171       return data
12172     end
12173   end
12174 end

```

(End of definition for get_luadata.)

55.5 Parsing Unicode data files

`__kernel_codepoint_data:wn` This provides a macro to access a Unicode data entry. The first argument is a unbraced number, the second a string identifying the kind of data to read. The supported kinds match the data tables supported by `__kernel_codepoint_data:nn` in other engines.

`__codepoint_nfd:w`

```
12175 do
12176   if get_luaadata then
12177     local saved_unidata = get_luaadata'lua-uni-data'
12178     if saved_unidata then saved_unidata() end
12179   end
12180   if register_luaadata then
12181     register_luaadata('lua-uni-data', function()
12182       return string.format("load(%q, nil, 'b')", string.dump(require'lua-uni-data-preload'.
12183     end)
12184   end
12185
12186   local uni_data = require'lua-uni-data'
12187   local tables, decomposition_mapping = uni_data.tables, uni_data.misc.decomposition_mapping
12188
12189   luacmd('__kernel_codepoint_data:wn', function()
12190     local cp = scan_int()
12191     local table = scan_argument()
12192     return sprint(-2, tostring(tables[table][cp]))
12193   end)
12194
12195   luacmd('__codepoint_nfd:w', function()
12196     local cp = scan_int()
12197     local decomposed = decomposition_mapping[cp]
12198     if decomposed then
12199       for i=1,2 do
12200         if decomposed[i] then
12201           sprint(-2, lbrace, tostring(decomposed[i]), rbrace)
12202         else
12203           sprint(-2, lbrace, rbrace)
12204         end
12205       end
12206     else
12207       return sprint(-2, lbrace, tostring(cp), rbrace, lbrace, rbrace)
12208     end
12209   end)
12210 end
```

(End of definition for `__kernel_codepoint_data:wn` and `__codepoint_nfd:w`.)

12211 `</lua>`

Chapter 56

13legacy implementation

12212 `<*code>`

12213 `<@@=legacy>`

`\legacy_if_p:n` A friendly wrapper. We need to use the `\if:w` approach here, rather than testing against `\iftrue`/`\iffalse` as the latter approach fails for primitive conditionals such as `\ifmmode`. The `\reverse_if:N` here means that we get a slightly more useful error if the name is undefined.

`\legacy_if:nTF`

```
12214 \prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
12215 {
12216   \exp_after:wN \reverse_if:N
12217   \cs:w if#1 \cs_end:
12218   \prg_return_false:
12219   \else:
12220   \prg_return_true:
12221   \fi:
12222 }
```

(End of definition for `\legacy_if:nTF`. This function is documented on page 111.)

`\legacy_if_set_true:n` A friendly wrapper.

`\legacy_if_set_false:n`

`\legacy_if_gset_true:n`

`\legacy_if_gset_false:n`

```
12223 \cs_new_protected:Npn \legacy_if_set_true:n #1
12224 { \cs_set_eq:cN { if#1 } \if_true: }
12225 \cs_new_protected:Npn \legacy_if_set_false:n #1
12226 { \cs_set_eq:cN { if#1 } \if_false: }
12227 \cs_new_protected:Npn \legacy_if_gset_true:n #1
12228 { \cs_gset_eq:cN { if#1 } \if_true: }
12229 \cs_new_protected:Npn \legacy_if_gset_false:n #1
12230 { \cs_gset_eq:cN { if#1 } \if_false: }
```

(End of definition for `\legacy_if_set_true:n` and others. These functions are documented on page 111.)

`\legacy_if_set:nn` A more elaborate wrapper.

`\legacy_if_gset:nn`

```
12231 \cs_new_protected:Npn \legacy_if_set:nn #1#2
12232 {
12233   \bool_if:nTF {#2} \legacy_if_set_true:n \legacy_if_set_false:n
12234   {#1}
12235 }
```

```

12236 \cs_new_protected:Npn \legacy_if_gset:nn #1#2
12237 {
12238   \bool_if:nTF {#2} \legacy_if_gset_true:n \legacy_if_gset_false:n
12239   {#1}
12240 }

```

(End of definition for \legacy_if_set:nn and \legacy_if_gset:nn. These functions are documented on page [111](#).)

```

12241 \</code>

```

Chapter 57

l3tl implementation

12242 `<*code>`

12243 `<@@=tl>`

A token list variable is a $\text{\TeX}$ macro that holds tokens. By using the $\varepsilon\text{-TeX}$ primitive `\unexpanded` inside a $\text{\TeX}$ `\edef` it is possible to store any tokens, including `#`, in this way.

57.1 Functions

`\__kernel_tl_set:Nx` These two are supplied to get better performance for macros which would otherwise use `\tl_set:Ne` or `\tl_gset:Ne` internally.

12244 `\cs_new_eq:NN \__kernel_tl_set:Nx \cs_set_nopar:Npe`

12245 `\cs_new_eq:NN \__kernel_tl_gset:Nx \cs_gset_nopar:Npe`

(End of definition for `\__kernel_tl_set:Nx` and `\__kernel_tl_gset:Nx`.)

`\tl_new:N` Creating new token list variables is a case of checking for an existing definition and doing the definition.

`\tl_new:c`

12246 `\cs_new_protected:Npn \tl_new:N #1`

12247 `{`

12248 `\__kernel_chk_if_free_cs:N #1`

12249 `\cs_gset_eq:NN #1 \c_empty_tl`

12250 `}`

12251 `\cs_generate_variant:Nn \tl_new:N { c }`

(End of definition for `\tl_new:N`. This function is documented on page 113.)

`\tl_const:Nn` Constants are also easy to generate. They use `\cs_gset_nopar:Npe` instead of `\__kernel_tl_gset:Nx` so that the correct scope checking for `c`, instead of for `g`, is applied when `\debug_on:n { check-declarations }` is used. Constant assignment functions are patched specially in `l3debug` to apply such checks.

`\tl_const:NV`

`\tl_const:Ne`

`\tl_const:Nx`

`\tl_const:cn`

`\tl_const:cV`

`\tl_const:ce`

`\tl_const:cx`

12252 `\cs_new_protected:Npn \tl_const:Nn #1#2`

12253 `{`

12254 `\__kernel_chk_if_free_cs:N #1`

12255 `\cs_gset_nopar:Npe #1 { \__kernel_exp_not:w {#2} }`

12256 `}`

12257 `\cs_generate_variant:Nn \tl_const:Nn { NV , Ne , c , cV , ce }`

12258 `\cs_generate_variant:Nn \tl_const:Nn { Nx , cx }`

(End of definition for `\tl_const:Nn`. This function is documented on page 114.)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear:c
\tl_gclear:N
\tl_gclear:c
12259 \cs_new_protected:Npn \tl_clear:N #1
12260 { \tex_let:D #1 = ~ \c_empty_tl }
12261 \cs_new_protected:Npn \tl_gclear:N #1
12262 { \tex_global:D \tex_let:D #1 ~ \c_empty_tl }
12263 \cs_generate_variant:Nn \tl_clear:N { c }
12264 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End of definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 114.)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c
12265 \cs_new_protected:Npn \tl_clear_new:N #1
12266 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
12267 \cs_new_protected:Npn \tl_gclear_new:N #1
12268 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
12269 \cs_generate_variant:Nn \tl_clear_new:N { c }
12270 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(End of definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 114.)

`\tl_set_eq:NN` For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit. In addition this ensures that a braced second argument will not cause problems.

```
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc
12271 \cs_new_protected:Npn \tl_set_eq:NN #1#2
12272 { \tex_let:D #1 = ~ #2 }
12273 \cs_new_protected:Npn \tl_gset_eq:NN #1#2
12274 { \tex_global:D \tex_let:D #1 = ~ #2 }
12275 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
12276 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
```

(End of definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 114.)

`\tl_concat:NNN` Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop:` would lead to problems later on.

```
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc
12277 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
12278 {
12279   \__kernel_tl_set:Nx #1
12280   {
12281     \__kernel_exp_not:w \exp_after:wN {#2}
12282     \__kernel_exp_not:w \exp_after:wN {#3}
12283   }
12284 }
12285 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
12286 {
12287   \__kernel_tl_gset:Nx #1
12288   {
12289     \__kernel_exp_not:w \exp_after:wN {#2}
12290     \__kernel_exp_not:w \exp_after:wN {#3}

```

```

12291     }
12292   }
12293   \cs_generate_variant:Nn \tl_concat:NNN { ccc }
12294   \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End of definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 114.)

`\tl_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\tl_if_exist_p:c` 12295 `\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\tl_if_exist:NTF` 12296 `\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\tl_if_exist:cTF`
 (End of definition for `\tl_if_exist:NTF`. This function is documented on page 114.)

57.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```

12297 \tl_const:Nn \c_empty_tl { }

```

(End of definition for `\c_empty_tl`. This variable is documented on page 130.)

`\c_novalue_tl` A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```

12298 \group_begin:
12299 \tex_catcode:D '- = 11 ~
12300 \tl_const:Ne \c_novalue_tl { - NoValue \token_to_str:N - }
12301 \group_end:

```

(End of definition for `\c_novalue_tl`. This variable is documented on page 130.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```

12302 \tl_const:Nn \c_space_tl { ~ }

```

(End of definition for `\c_space_tl`. This variable is documented on page 131.)

57.3 Adding to token list variables

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by `TEX` more or less redundant. The `\tl_set:No` version is done by hand as it is used quite a lot.

```

\tl_set:Nn 12303 \cs_new_protected:Npn \tl_set:Nn #1#2
\tl_set:Nv 12304 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w {#2} } }
\tl_set:Ne 12305 \cs_new_protected:Npn \tl_set:Ne #1#2
\tl_set:Nf 12306 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
\tl_set:Nx 12307 \cs_new_protected:Npn \tl_gset:Nn #1#2
\tl_set:cn 12308 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w {#2} } }
\tl_set:cV 12309 \cs_new_protected:Npn \tl_gset:No #1#2
\tl_set:cv 12310 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
\tl_set:co 12311 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Ne , Nf }
\tl_set:ce 12312 \cs_generate_variant:Nn \tl_set:Nn { c , cV , cv , ce , cf }
\tl_set:cf 12313 \cs_generate_variant:Nn \tl_set:No { c }
\tl_set:cx 12314 \cs_generate_variant:Nn \tl_set:Nn { Nx , cx }
\tl_gset:Nn 12315 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Ne , Nf }

```

```

\tl_gset:Nv
\tl_gset:Nv
\tl_gset:No
\tl_gset:Ne
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cn
\tl_gset:cV
\tl_gset:cv
\tl_gset:co

```

```

12316 \cs_generate_variant:Nn \tl_gset:Nn { c, cV , cv , ce , cf }
12317 \cs_generate_variant:Nn \tl_gset:No { c }
12318 \cs_generate_variant:Nn \tl_gset:Nn { Nx , cx }

```

(End of definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 114.)

```

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.
\tl_put_left:NV 12319 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:Nv 12320 {
\tl_put_left:Ne 12321   \__kernel_tl_set:Nx #1
\tl_put_left:No 12322   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_put_left:Nx 12323 }
\tl_put_left:cn 12324 \cs_new_protected:Npn \tl_put_left:Nv #1#2
\tl_put_left:cV 12325 {
\tl_put_left:cv 12326   \__kernel_tl_set:Nx #1
\tl_put_left:ce 12327   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_put_left:co 12328 }
\tl_put_left:cx 12329 \cs_new_protected:Npn \tl_put_left:Nv #1#2
\tl_gput_left:Nn 12330 {
\tl_gput_left:NV 12331   \__kernel_tl_set:Nx #1
\tl_gput_left:Nv 12332   { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
\tl_gput_left:Ne 12333 }
\tl_gput_left:No 12334 \cs_new_protected:Npn \tl_gput_left:Ne #1#2
\tl_gput_left:Nx 12335 {
\tl_gput_left:cn 12336   \__kernel_tl_set:Nx #1
\tl_gput_left:cV 12337   {
12338     \__kernel_exp_not:w \tex_expanded:D { {#2} }
12339     \__kernel_exp_not:w \exp_after:wN {#1}
12340   }
\tl_gput_left:ce 12341 }
\tl_gput_left:co 12342 \cs_new_protected:Npn \tl_gput_left:No #1#2
\tl_gput_left:cx 12343 {
12344   \__kernel_tl_set:Nx #1
12345   {
12346     \__kernel_exp_not:w \exp_after:wN {#2}
12347     \__kernel_exp_not:w \exp_after:wN {#1}
12348   }
12349 }
12350 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
12351 {
12352   \__kernel_tl_gset:Nx #1
12353   { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12354 }
12355 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
12356 {
12357   \__kernel_tl_gset:Nx #1
12358   { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
12359 }
12360 \cs_new_protected:Npn \tl_gput_left:Nv #1#2
12361 {
12362   \__kernel_tl_gset:Nx #1
12363   { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12364 }
12365 \cs_new_protected:Npn \tl_gput_left:Ne #1#2

```

```

12366 {
12367   \__kernel_tl_gset:Nx #1
12368   {
12369     \__kernel_exp_not:w \tex_expanded:D { {#2} }
12370     \__kernel_exp_not:w \exp_after:wN {#1}
12371   }
12372 }
12373 \cs_new_protected:Npn \tl_gput_left:No #1#2
12374 {
12375   \__kernel_tl_gset:Nx #1
12376   {
12377     \__kernel_exp_not:w \exp_after:wN {#2}
12378     \__kernel_exp_not:w \exp_after:wN {#1}
12379   }
12380 }
12381 \cs_generate_variant:Nn \tl_put_left:Nn { c }
12382 \cs_generate_variant:Nn \tl_put_left:Nv { c }
12383 \cs_generate_variant:Nn \tl_put_left:Nv { c }
12384 \cs_generate_variant:Nn \tl_put_left:Ne { c }
12385 \cs_generate_variant:Nn \tl_put_left:No { c }
12386 \cs_generate_variant:Nn \tl_put_left:Nn { Nx, cx }
12387 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
12388 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
12389 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
12390 \cs_generate_variant:Nn \tl_gput_left:Ne { c }
12391 \cs_generate_variant:Nn \tl_gput_left:No { c }
12392 \cs_generate_variant:Nn \tl_gput_left:Nn { Nx, cx }

```

(End of definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 114.)

`\tl_put_right:Nn` The same on the right.

```

\tl_put_right:Nv 12393 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:Nv 12394 { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\tl_put_right:Nv 12395 \cs_new_protected:Npn \tl_put_right:Nv #1#2
\tl_put_right:Ne 12396 {
\tl_put_right:No 12397   \__kernel_tl_set:Nx #1
\tl_put_right:Nx 12398   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
\tl_put_right:cn 12399 }
\tl_put_right:cV 12400 \cs_new_protected:Npn \tl_put_right:Nv #1#2
\tl_put_right:cv 12401 {
\tl_put_right:ce 12402   \__kernel_tl_set:Nx #1
\tl_put_right:co 12403   { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
\tl_put_right:cx 12404 }
\tl_gput_right:Nn 12405 \cs_new_protected:Npn \tl_put_right:Ne #1#2
\tl_gput_right:Nv 12406 {
\tl_gput_right:Nv 12407   \__kernel_tl_set:Nx #1
\tl_gput_right:Ne 12408   {
\tl_gput_right:No 12409     \__kernel_exp_not:w \exp_after:wN {#1}
\tl_gput_right:Nx 12410     \__kernel_exp_not:w \tex_expanded:D { {#2} }
\tl_gput_right:cn 12411   }
\tl_gput_right:cV 12412 }
\tl_gput_right:cv 12413 \cs_new_protected:Npn \tl_put_right:No #1#2
\tl_gput_right:ce 12414 {
\tl_gput_right:co
\tl_gput_right:cx

```

```

12415     \__kernel_tl_set:Nx #1
12416     {
12417         \__kernel_exp_not:w \exp_after:wN {#1}
12418         \__kernel_exp_not:w \exp_after:wN {#2}
12419     }
12420 }
12421 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
12422 { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
12423 \cs_new_protected:Npn \tl_gput_right:Nv #1#2
12424 {
12425     \__kernel_tl_gset:Nx #1
12426     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
12427 }
12428 \cs_new_protected:Npn \tl_gput_right:Nv #1#2
12429 {
12430     \__kernel_tl_gset:Nx #1
12431     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
12432 }
12433 \cs_new_protected:Npn \tl_gput_right:Ne #1#2
12434 {
12435     \__kernel_tl_gset:Nx #1
12436     {
12437         \__kernel_exp_not:w \exp_after:wN {#1}
12438         \__kernel_exp_not:w \tex_expanded:D { {#2} }
12439     }
12440 }
12441 \cs_new_protected:Npn \tl_gput_right:No #1#2
12442 {
12443     \__kernel_tl_gset:Nx #1
12444     {
12445         \__kernel_exp_not:w \exp_after:wN {#1}
12446         \__kernel_exp_not:w \exp_after:wN {#2}
12447     }
12448 }
12449 \cs_generate_variant:Nn \tl_put_right:Nn { c }
12450 \cs_generate_variant:Nn \tl_put_right:Nv { c }
12451 \cs_generate_variant:Nn \tl_put_right:Nv { c }
12452 \cs_generate_variant:Nn \tl_put_right:Ne { c }
12453 \cs_generate_variant:Nn \tl_put_right:No { c }
12454 \cs_generate_variant:Nn \tl_put_right:Nn { Nx , cx }
12455 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
12456 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
12457 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
12458 \cs_generate_variant:Nn \tl_gput_right:Ne { c }
12459 \cs_generate_variant:Nn \tl_gput_right:No { c }
12460 \cs_generate_variant:Nn \tl_gput_right:Nn { Nx, cx }

```

(End of definition for `\tl_put_right:Nn` and `\tl_gput_right:Nn`. These functions are documented on page 115.)

57.4 Internal quarks and quark-query functions

`\q__tl_nil` Internal quarks.
`\q__tl_mark`
`\q__tl_stop`


```

12461 \quark_new:N \q__tl_nil
12462 \quark_new:N \q__tl_mark
12463 \quark_new:N \q__tl_stop

```

(End of definition for `\q__tl_nil`, `\q__tl_mark`, and `\q__tl_stop`.)

```

\__tl_recursion_tail Internal recursion quarks.
\__tl_recursion_stop 12464 \quark_new:N \q__tl_recursion_tail
12465 \quark_new:N \q__tl_recursion_stop

```

(End of definition for `\__tl_recursion_tail` and `\q__tl_recursion_stop`.)

```

\__tl_if_recursion_tail_break:n Functions to query recursion quarks.
\__tl_if_recursion_tail_stop:p:n 12466 \__kernel_quark_new_test:N \__tl_if_recursion_tail_break:nN
\__tl_if_recursion_tail_stop:nTF 12467 \__kernel_quark_new_conditional:Nn \__tl_quark_if_nil:n { TF }

```

(End of definition for `\__tl_if_recursion_tail_break:nN` and `\__tl_if_recursion_tail_stop:nTF`.)

57.5 Reassigning token list category codes

`\c__tl_rescan_marker_tl` The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```

12468 \tl_const:Ne \c__tl_rescan_marker_tl { : \token_to_str:N : }

```

(End of definition for `\c__tl_rescan_marker_tl`.)

```

\tl_set_rescan:Nnn In a group, after some initial setup explained below and the user setup #3 (followed by
\tl_set_rescan:NnV \scan_stop: to be safe), there is a call to \__tl_set_rescan:nNN. This shared auxiliary
\tl_set_rescan:Nne defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-
\tl_set_rescan:Nno line files, it calls (with the same arguments) \__tl_set_rescan_multi:nNN, whose code
\tl_set_rescan:Nnx is included here to help understand the approach. This function rescans its argument #1,
\tl_set_rescan:cnm closes the group, and performs the assignment.
\tl_set_rescan:cnV
\tl_set_rescan:cne
\tl_set_rescan:cno
\tl_set_rescan:cnx

```

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a TeX error occurs:

```
! File ended while scanning definition of ...
```

```

\tl_gset_rescan:Nnn A related minor issue is a warning due to opening a group before the \scantokens and
\tl_gset_rescan:NnV closing it inside that temporary file; we avoid that by setting \tracingnesting. The
\tl_gset_rescan:Nne standard solution to the “File ended” error is to grab the rescanned tokens as a delimited
\tl_gset_rescan:Nno argument of an auxiliary, here \__tl_rescan:NNw, that performs the assignment, then let
\tl_gset_rescan:Nnx TeX “execute” the end of file marker. As usual in delimited arguments we use \prg_do_-
\tl_gset_rescan:cnm nothing: to avoid stripping an outer set braces: this is removed by using o-expanding
\tl_gset_rescan:cnV assignments. The delimiter cannot appear within the rescanned token list because it
\tl_gset_rescan:cne contains twice the same character, with different catcodes.
\tl_gset_rescan:cno
\tl_gset_rescan:cnx

```

For `\tl_rescan:nn` we cannot simply call `\__tl_set_rescan:NNnn \prg_do_-nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become

```

\__tl_rescan_aux:
\__tl_set_rescan:NNnn
\__tl_set_rescan_multi:nNN
\__tl_rescan:NNw

```

`\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to `-1` if it was `32` (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally f-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in e-type arguments when `\expanded` is not available.

```

12469 \cs_new_protected:Npn \tl_rescan:nn #1#2
12470 {
12471   \tl_set_rescan:Nnn \l__tl_tmpa_tl {#1} {#2}
12472   \exp_after:wN \__tl_rescan_aux:
12473   \l__tl_tmpa_tl
12474 }
12475 \cs_generate_variant:Nn \tl_rescan:nn { nV }
12476 \exp_args:NNo \cs_new_protected:Npn \__tl_rescan_aux:
12477 { \tl_clear:N \l__tl_tmpa_tl }
12478 \cs_new_protected:Npn \tl_set_rescan:Nnn
12479 { \__tl_set_rescan:NNnn \tl_set:No }
12480 \cs_new_protected:Npn \tl_gset_rescan:Nnn
12481 { \__tl_set_rescan:NNnn \tl_gset:No }
12482 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
12483 {
12484   \group_begin:
12485   \if_false: { \fi:
12486     \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
12487     \int_compare:nNnT \tex_endlinechar:D = { 32 }
12488     { \int_set:Nn \tex_endlinechar:D { -1 } }
12489     \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
12490     #3 \scan_stop:
12491     \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
12492   \if_false: } \fi:
12493 }
12494 \cs_new_protected:Npn \__tl_set_rescan_multi:NNn #1#2#3
12495 {
12496   \tex_veryeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
12497   \exp_after:wN \__tl_rescan:NNw
12498   \exp_after:wN #2
12499   \exp_after:wN #3
12500   \exp_after:wN \prg_do_nothing:
12501   \tex_scantokens:D {#1}
12502 }
12503 \exp_args:Nno \use:nn
12504 { \cs_new:Npn \__tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl
12505 {
12506   \group_end:
12507   #1 #2 {#3}
12508 }
12509 \cs_generate_variant:Nn \tl_set_rescan:Nnn { NnV , Nne , c , cnV , cne }
12510 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx , cno , cnx }

```

```

12511 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { NnV , Nne , c , cnV , cne }
12512 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx , cno , cnx }

```

(End of definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 129.)

```

\__tl_set_rescan:nNN
\__tl_set_rescan_single:nnNN
\__tl_set_rescan_single_aux:nnnNN
\__tl_set_rescan_single_aux:w

```

The function `\__tl_set_rescan:nNN` calls `\__tl_set_rescan_multi:nNN` or `\__tl_set_rescan_single:nnNN { ' }` depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as `TEX` removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, `11` (letter) and `12` (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `\__tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `\__tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `\__tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c__tl_set_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof to ::{<code1>}'::{<code2>}` `\s__tl_stop`. The auxiliary `\__tl_set_rescan_single:NNww` runs the `o`-expanding assignment, expanding either `<code1>` or `<code2>` before its the main argument `#3`. In the typical case without comment character, `<code1>` is expanded, removing the leading `'`. In the rarer case with comment character, `<code2>` is expanded, calling `\__tl_set_rescan_single_aux:w`, which removes the trailing `::{<code1>}` and the leading `'`.

```

12513 \cs_new_protected:Npn \__tl_set_rescan:nNN #1
12514 {
12515   \int_compare:nNnTF \tex_newlinechar:D < 0
12516   { \use_ii:nn }
12517   {
12518     \exp_args:Nnf \tl_if_in:nnTF {#1}
12519     { \char_generate:nn { \tex_newlinechar:D } { 12 } }
12520   }
12521   { \__tl_set_rescan_multi:nNN }
12522   {
12523     \int_set:Nn \tex_endlinechar:D { -1 }
12524     \__tl_set_rescan_single:nnNN { ' ' }
12525   }
12526   {#1}
12527 }
12528 \cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1

```

```

12529 {
12530   \int_compare:nNnTF
12531     { \char_value_catcode:n {#1} / 2 } = 6
12532     {
12533       \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
12534         \c__tl_rescan_marker_tl
12535         { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
12536     }
12537     {
12538       \int_compare:nNnTF {#1} < { '\~ }
12539       {
12540         \exp_args:Nf \__tl_set_rescan_single:nnNN
12541         { \int_eval:n { #1 + 1 } }
12542       }
12543       { \__tl_set_rescan_multi:nNN }
12544     }
12545   }
12546   \cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
12547   {
12548     \tex_everyeof:D
12549     {
12550       #1 \use_none:n
12551       #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
12552       \s__tl_stop
12553     }
12554     \cs_set:Npn \__tl_rescan:NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
12555     {
12556       \group_end:
12557       ##1 ##2 { ##4 ##3 }
12558     }
12559     \exp_after:wN \__tl_rescan:NNw
12560     \exp_after:wN #4
12561     \exp_after:wN #5
12562     \tex_scantokens:D { #2 #3 #2 }
12563   }
12564   \exp_args:Nno \use:nn
12565   { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
12566   \c__tl_rescan_marker_tl #2
12567   { \use_i:nn \exp_end: #1 }

```

(End of definition for `\__tl_set_rescan:nNN` and others.)

`\tl_set_rescan:Nnn`

`\tl_gset_rescan:Nnn`

`\__tl_set_rescan_single:nnNN`

`\__tl_set_rescan_single_aux:nnNN`

`\__tl_set_rescan_aux:w`

`\__tl_set_rescan_multi:nNN`

`\__tl_set_rescan_multi_aux:nNN`

If the `\scantextokens` is available, the above is not all required: in particular, in LuaMetaTeX there is a statement to forget about `\scantokens` so it seems sensible to use `\scantextokens` if possible. We keep as much of the shared code path, but in this case `\everyeof` is unused (and we do not have to worry about end-of-file errors). Getting the documented behavior for single versus multiple line input requires a bit of work; for the latter we need to add back in the `\endlinechar` that is deliberately omitted. That is a bit awkward as the engine really doesn't want to do this!

```

12568 \cs_if_exist:NT \tex_scantextokens:D
12569 {
12570   \cs_gset_protected:Npn \tl_set_rescan:Nnn
12571     { \__tl_set_rescan:NNnn \tl_set:Nn }
12572   \cs_gset_protected:Npn \tl_gset_rescan:Nnn

```

```

12573 { \_tl_set_rescan:NNnn \tl_gset:Nn }
12574 \cs_gset_protected:Npn \_tl_set_rescan_single:nnNN #1
12575 {
12576   \int_compare:nNnTF
12577   { \char_value_catcode:n {#1} / 2 } = 6
12578   {
12579     \exp_args:Nf \_tl_set_rescan_single_aux:nnNN
12580     { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
12581   }
12582   {
12583     \int_compare:nNnTF {#1} < { '\~ }
12584     {
12585       \exp_args:Nf \_tl_set_rescan_single:nnNN
12586       { \int_eval:n { #1 + 1 } }
12587     }
12588     { \_tl_set_rescan_multi:nnN }
12589   }
12590 }
12591 \cs_new_protected:Npn \_tl_set_rescan_aux:w #1
12592 {
12593   \cs_new_protected:Npn \_tl_set_rescan_single_aux:nnNN ##1##2##3##4
12594   {
12595     \tl_set:No \l__tl_tmpa_tl { \tex_scantextokens:D { ##1 ##2 ##1 } }
12596     \cs_set_protected:Npn \_tl_set_rescan:w
12597     ###1 ##1 #1 ####2 #1 ####3 \s__tl_stop
12598     {
12599       \tl_if_blank:nTF {####3}
12600       { \_tl_set_rescan_aux:w ####1 \s__tl_stop }
12601       { \tl_set:No \l__tl_tmpa_tl { \use_none:n ####1 } }
12602     }
12603     \exp_after:wN \_tl_set_rescan:w \l__tl_tmpa_tl #1 ##1 #1 #1 \s__tl_stop
12604     \exp_args:NNNo \group_end:
12605     ##3 ##4 \l__tl_tmpa_tl
12606   }
12607   \cs_gset_protected:Npn \_tl_set_rescan_aux:w ##1 #1 ##2 \s__tl_stop
12608   { \tl_set:No \l__tl_tmpa_tl { \use_none:n ##1 } }
12609 }
12610 \exp_args:NV \_tl_set_rescan_aux:w \c_tl_rescan_marker_tl
12611 \cs_gset_protected:Npn \_tl_set_rescan_multi:nnN #1#2#3
12612 {
12613   \str_if_eq:eeTF { \tl_head:e { \tl_reverse:n {#1} } }
12614   { \char_generate:nn { \tex_endlinechar:D } { 12 } }
12615   { \_tl_set_rescan_multi_aux:nnN {#1} }
12616   {
12617     \exp_args:Ne \_tl_set_rescan_multi_aux:nnN
12618     { #1 \char_generate:nn { \tex_endlinechar:D } { 12 } }
12619   }
12620   #2 #3
12621 }
12622 \cs_new_protected:Npn \_tl_set_rescan_multi_aux:nnN #1#2#3
12623 {
12624   \tex_expanded:D
12625   {
12626     \exp_not:N \exp_args:NNNo \group_end:

```

```

12627         #2 \exp_not:N #3
12628         {
12629             \exp_not:N \tex_scantextokens:D
12630             {
12631                 #1
12632                 \char_generate:nn { 13 } { 12 }
12633             }
12634         }
12635     }
12636 }
12637 }

```

(End of definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 129.)

`\tl_retokenize:n` A thin wrapper from `\scantokens`. At present, we are told by Hans that LuaMetaTeX will retain this primitive, so we should be OK for all engines.

`\tl_retokenize:V`

```

12638 \group_begin:
12639   \tex_catcode:D { ^M = 12 \scan_stop: %
12640   \cs_new_protected:Npn \tl_retokenize:n #1 %
12641   { %
12642     \group_begin: %
12643     \int_set:Nn \tex_newlinechar:D { ^J } %
12644     \int_set:Nn \tex_endlinechar:D { ^M } %
12645     \exp_after:wN \group_end: %
12646     \tex_scantokens:D \exp_after:wN %
12647     { \tl_to_str:n {#1} } %
12648   } %
12649 \group_end: %
12650 \cs_generate_variant:Nn \tl_retokenize:n { V }

```

(End of definition for `\tl_retokenize:n`. This function is documented on page 130.)

57.6 Modifying token list variables

`\tl_replace_once:Nnn`

`\tl_replace_once:NVn`

`\tl_replace_once:NnV`

`\tl_replace_once:Nen`

`\tl_replace_once:Nne`

`\tl_replace_once:Nee`

`\tl_replace_once:Nxn`

`\tl_replace_once:Nnx`

`\tl_replace_once:cnn`

`\tl_replace_once:cVn`

`\tl_replace_once:cnV`

`\tl_replace_once:cen`

`\tl_replace_once:cne`

`\tl_replace_once:cee`

`\tl_replace_once:cxn`

`\tl_replace_once:cnx`

`\tl_replace_once:cxx`

`\tl_replace_once:cnn`

`\tl_replace_once:cVn`

`\tl_replace_once:cnV`

`\tl_replace_once:cen`

`\tl_replace_once:cne`

`\tl_replace_once:cee`

`\tl_replace_once:cxn`

`\tl_replace_once:cnx`

`\tl_replace_once:cxx`

`\tl_replace_once:cnn`

All of the `replace` functions call `\__tl_replace:NnnNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`\__tl_replace_next:w`) or stops (`\__tl_replace_wrap:w`) after the first replacement. Next comes an e-type assignment function `\tl_set:Ne` or `\tl_gset:Ne` for local or global replacements. Finally, the three arguments `<tl var>` `{<pattern>}` `{<replacement>}` provided by the user. When describing the auxiliary functions below, we denote the contents of the `<tl var>` by `<token list>`.

```

12651 \cs_new_protected:Npn \tl_replace_once:Nnn
12652 { \__tl_replace:NnnNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }
12653 \cs_new_protected:Npn \tl_replace_once:NnV
12654 { \__tl_replace:NnnNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
12655 \cs_new_protected:Npn \tl_replace_all:Nnn
12656 { \__tl_replace:NnnNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx }
12657 \cs_new_protected:Npn \tl_replace_all:Nnn
12658 { \__tl_replace:NnnNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }
12659 \cs_generate_variant:Nn \tl_replace_once:Nnn
12660 { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12661 \cs_generate_variant:Nn \tl_replace_once:Nnn
12662 { Nx , Nnx , Nxx , cxn , cnx , cxx }

```

```

12663 \cs_generate_variant:Nn \tl_greplace_once:Nnn
12664 { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12665 \cs_generate_variant:Nn \tl_greplace_once:Nnn
12666 { Nx , Nnx , Nxx , cxn , cnx , cxx }
12667 \cs_generate_variant:Nn \tl_replace_all:Nnn
12668 { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12669 \cs_generate_variant:Nn \tl_replace_all:Nnn
12670 { Nx , Nnx , Nxx , cxn , cnx , cxx }
12671 \cs_generate_variant:Nn \tl_greplace_all:Nnn
12672 { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
12673 \cs_generate_variant:Nn \tl_greplace_all:Nnn
12674 { Nx , Nnx , Nxx , cxn , cnx , cxx }

```

(End of definition for `\tl_replace_once:Nnn` and others. These functions are documented on page 127.)

```

\__tl_replace:NnnNnnNnn
\__tl_replace_auxi:NnnNnnNnn
\__tl_replace_auxii:NnnNnnNnn
\__tl_replace_next:w
\__tl_replace_next_aux:w
\__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `\__tl_replace_auxii:NnnNnnNnn` we need a `<delimiter>` with the following properties:

- all occurrences of the `<pattern>` #6 in “`<token list> <delimiter>`” belong to the `<token list>` and have no overlap with the `<delimiter>`,
- the first occurrence of the `<delimiter>` in “`<token list> <delimiter>`” is the trailing `<delimiter>`.

We first find the building blocks for the `<delimiter>`, namely two tokens `<A>` and `<B>` such that `<A>` does not appear in #6 and #6 is not `<B>` (this condition is trivial if #6 has more than one token). Then we consider the delimiters “`<A>`” and “`<A> <A>n <B> <A>n <B>`”, for $n \geq 1$, where `<A>n` denotes n copies of `<A>`, and we choose as our `<delimiter>` the first one which is not in the `<token list>`.

Every delimiter in the set obeys the first condition: #6 does not contain `<A>` hence cannot be overlapping with the `<token list>` and the `<delimiter>`, and it cannot be within the `<delimiter>` since it would have to be in one of the two `<B>` hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the `<delimiter>` we choose does not appear in the `<token list>`. Additionally, the set of delimiters is such that a `<token list>` of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a `<delimiter>` with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “`<A>`” in the list of delimiters to try, so that the `<delimiter>` is simply `\q__tl_mark` in the most common situation where neither the `<token list>` nor the `<pattern>` contains `\q__tl_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty `<pattern>` #6 is an error, and if #1 is absent from both the `<token list>` #5 and the `<pattern>` #6 then we can use it as the `<delimiter>` through `\__tl_replace_auxii:NnnNnnNnn {#1}`. Otherwise, we end up calling `\__tl_replace:NnnNnnNnn` repeatedly with the first two arguments `\q__tl_mark {?}`, `\? {??}`, `\?? {???}`, and so on, until #6 does not contain the control sequence #1, which we take as our `<A>`. The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose `<B>` to be `\q__tl_nil` or `\q__tl_stop` such that it is not equal to #6.

The `\__tl_replace_auxi:NnnNnnNnn` auxiliary receives `{<A>}` and `{<A>n<B>}` as its arguments, initially with $n = 1$. If “`<A> <A>n<B> <A>n<B>`” is in the `<token list>` then

increase n and try again. Once it is not anymore in the $\langle token\ list \rangle$ we take it as our $\langle delimiter \rangle$ and pass this to the `auxii` auxiliary.

```

12675 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
12676 {
12677   \tl_if_empty:nTF {#6}
12678   {
12679     \msg_error:nne { kernel } { empty-search-pattern }
12680     { \tl_to_str:n {#7} }
12681   }
12682   {
12683     \tl_if_in:ontF { #5 #6 } {#1}
12684     {
12685       \tl_if_in:nnTF {#6} {#1}
12686       { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
12687       {
12688         \__tl_quark_if_nil:nTF {#6}
12689         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_stop } }
12690         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_nil } }
12691       }
12692     }
12693     { \__tl_replace_auxii:nNNNnn {#1} }
12694     #3#4#5 {#6} {#7}
12695   }
12696 }
12697 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
12698 {
12699   \tl_if_in:NnTF #1 { #2 #3 #3 }
12700   { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
12701   { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
12702 }

```

The auxiliary `\__tl_replace_auxii:nNNNnn` receives the following arguments:

$\{\langle delimiter \rangle\}$ $\langle function \rangle$ $\langle assignment \rangle$
 $\langle tl\ var \rangle$ $\{\langle pattern \rangle\}$ $\{\langle replacement \rangle\}$

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 {...}`, an e-expanding $\langle assignment \rangle$ `#3` to the $\langle tl\ var \rangle$ `#4`. The auxiliary `\__tl_replace_next:w` is called, followed by the $\langle token\ list \rangle$, some tokens including the $\langle delimiter \rangle$ `#1`, followed by the $\langle pattern \rangle$ `#5`. This auxiliary finds an argument delimited by `#5` (the presence of a trailing `#5` avoids runaway arguments) and calls `\__tl_replace_wrap:w` to test whether this `#5` is found within the $\langle token\ list \rangle$ or is the trailing one.

If on the one hand it is found within the $\langle token\ list \rangle$, then `##1` cannot contain the $\langle delimiter \rangle$ `#1` that we worked so hard to obtain, thus `\__tl_replace_wrap:w` gets `##1` as its own argument `##1`, and protects it against the e-expanding assignment. It also finds `\exp_not:n` as `##2` and does nothing to it, thus letting through `\exp_not:n` $\{\langle replacement \rangle\}$ into the assignment. Note that `\__tl_replace_next:w` and `\__tl_replace_wrap:w` are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `\__tl_replace_next:w` is called to repeat the replacement, or `\__tl_replace_wrap:w` if we only want a single replacement. In this second case, `##1`

is the *<remaining tokens>* in the *<token list>* and ##2 is some *<ending code>* which ends the assignment and removes the trailing tokens #5 using some `\if_false: { \fi: }` trickery because #5 may contain any delimiter.

If on the other hand the argument ##1 of `\__tl_replace_next:w` is delimited by the trailing *<pattern>* #5, then ##1 is “{ } { } *<token list>* *<delimiter>* { *<ending code>* }”, hence `\__tl_replace_wrap:w` finds “{ } { } *<token list>*” as ##1 and the *<ending code>* as ##2. It leaves the *<token list>* into the assignment and unbraces the *<ending code>* which removes what remains (essentially the *<delimiter>* and *<replacement>*).

```

12703 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
12704 {
12705   \group_align_safe_begin:
12706   \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
12707     { \__kernel_exp_not:w \exp_after:wN { \use_none:nn ##1 } ##2 }
12708   \cs_set:Npe \__tl_replace_next:w ##1 #5
12709     {
12710       \exp_not:N \__tl_replace_wrap:w ##1
12711       \exp_not:n { #1 }
12712       \exp_not:n { \exp_not:n {#6} }
12713       \exp_not:n { #2 { } { } }
12714     }
12715   #3 #4
12716   {
12717     \exp_after:wN \__tl_replace_next_aux:w
12718     #4
12719     #1
12720     {
12721       \if_false: { \fi: }
12722       \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12723     }
12724     #5
12725   }
12726   \group_align_safe_end:
12727 }
12728 \cs_new:Npn \__tl_replace_next_aux:w { \__tl_replace_next:w { } { } }
12729 \cs_new_eq:NN \__tl_replace_wrap:w ?
12730 \cs_new_eq:NN \__tl_replace_next:w ?

```

(End of definition for `\__tl_replace:NnNNNnn` and others.)

```

\tl_regex_replace_once:Nnn
\tl_regex_replace_once:cnn
\tl_regex_replace_once:NNn
\tl_regex_replace_once:cNn
\tl_regex_greplace_once:Nnn
\tl_regex_greplace_once:cnn
\tl_regex_greplace_once:NNn
\tl_regex_greplace_once:cNn
\tl_regex_replace_all:Nnn
\tl_regex_replace_all:cnn
\tl_regex_replace_all:NNn
\tl_regex_replace_all:cNn
\tl_regex_greplace_all:Nnn
\tl_regex_greplace_all:cnn
\tl_regex_greplace_all:NNn
\tl_regex_greplace_all:cNn

```

Wrappers.

```

12731 \cs_new_protected:Npn \tl_regex_replace_once:Nnn #1#2#3
12732   { \regex_replace_once:nnN {#2} {#3} #1 }
12733 \cs_generate_variant:Nn \tl_regex_replace_once:Nnn { c }
12734 \cs_new_protected:Npn \tl_regex_replace_once:NNn #1#2#3
12735   { \regex_replace_once:NnN #2 {#3} #1 }
12736 \cs_generate_variant:Nn \tl_regex_replace_once:NNn { c }
12737 \cs_new_protected:Npn \tl_regex_replace_all:Nnn #1#2#3
12738   { \regex_replace_all:nnN {#2} {#3} #1 }
12739 \cs_generate_variant:Nn \tl_regex_replace_all:Nnn { c }
12740 \cs_new_protected:Npn \tl_regex_replace_all:NNn #1#2#3
12741   { \regex_replace_all:NnN #2 {#3} #1 }
12742 \cs_generate_variant:Nn \tl_regex_replace_all:NNn { c }
12743 \group_begin:

```

```

12744 \cs_set_protected:Npn \__tl_tmp:w #1#2#3
12745 {
12746   \cs_new_protected:cpe { tl_regex_greplace_ #1 :N #2 n } ##1##2##3
12747   {
12748     \group_begin:
12749       \tl_set_eq:NN \exp_not:N \l__tl_tmpa_tl ##1
12750       \exp_not:c { regex_replace_ #1 :Nn #2 }
12751       #3 {##2} {##3} \exp_not:N \l__tl_tmpa_tl
12752       \tl_gset_eq:NN ##1 \exp_not:N \l__tl_tmpa_tl
12753     \group_end:
12754   }
12755   \cs_generate_variant:cn { tl_regex_greplace_ #1 :N #2 n } { c }
12756 }
12757 \__tl_tmp:w { once } n { }
12758 \__tl_tmp:w { once } N \use:n
12759 \__tl_tmp:w { all } n { }
12760 \__tl_tmp:w { all } N \use:n
12761 \group_end:

```

(End of definition for `\tl_regex_replace_once:Nnn` and others. These functions are documented on page 128.)

\tl_remove_once:Nn Removal is just a special case of replacement.

```

\tl_remove_once:NV 12762 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
\tl_remove_once:Ne 12763 { \tl_replace_once:Nnn #1 {#2} { } }
\tl_remove_once:cn 12764 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
\tl_remove_once:cV 12765 { \tl_greplace_once:Nnn #1 {#2} { } }
\tl_remove_once:ce 12766 \cs_generate_variant:Nn \tl_remove_once:Nn { NV , Ne , c , cV , ce }
\tl_gremove_once:Nn 12767 \cs_generate_variant:Nn \tl_gremove_once:Nn { NV , Ne , c , cV , ce }

```

\tl_gremove_once:Nn (End of definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 128.)

\tl_remove_all:Nn Removal is just a special case of replacement.

```

\tl_remove_all:NV 12768 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_remove_all:Ne 12769 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_remove_all:Nx 12770 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
\tl_remove_all:cn 12771 { \tl_greplace_all:Nnn #1 {#2} { } }
\tl_remove_all:cV 12772 \cs_generate_variant:Nn \tl_remove_all:Nn { NV , Ne , c , cV , ce }
\tl_remove_all:ce 12773 \cs_generate_variant:Nn \tl_remove_all:Nn { Nx , cx }
\tl_remove_all:cx 12774 \cs_generate_variant:Nn \tl_gremove_all:Nn { NV , Ne , c , cV , ce }
\tl_gremove_all:Nn 12775 \cs_generate_variant:Nn \tl_gremove_all:Nn { Nx , cx }

```

\tl_gremove_all:Nn (End of definition for `\tl_remove_all:Nn` and `\tl_gremove_all:Nn`. These functions are documented on page 129.)

\tl_remove_all:Nx
\tl_remove_all:cn
\tl_remove_all:cV

57.7 Token list conditionals

\tl_remove_all:ce These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

\tl_if_empty_p:N 12776 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\tl_gremove_all:cx 12777 {
\tl_if_empty_p:c 12778   \if_meaning:w #1 \c_empty_tl
\tl_if_empty:cTF 12779   \prg_return_true:

```

```

12780     \else:
12781         \prg_return_false:
12782     \fi:
12783 }
12784 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
12785 { c } { p , T , F , TF }

```

(End of definition for `\tl_if_empty:NTF`. This function is documented on page 115.)

`\tl_if_empty_p:n` The `\if:w` triggers the expansion of `\tl_to_str:n` which converts the argument to a string: this is empty if and only if the argument is. Then `\if:w \scan_stop: ... \scan_stop:` is true if and only if the string ... is empty. It could be tempting to use `\tl_if_empty:nTF` `\if:w \scan_stop: #1 \scan_stop:` directly. But this fails on a token list expanding to anything starting with `\scan_stop:` leaving everything that follows in the input stream.

```

12786 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
12787 {
12788     \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
12789         \prg_return_true:
12790     \else:
12791         \prg_return_false:
12792     \fi:
12793 }
12794 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
12795 { V , e } { p , TF , T , F }

```

(End of definition for `\tl_if_empty:nTF`. This function is documented on page 115.)

`\tl_if_empty_p:o` The auxiliary function `\__tl_if_empty_if:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:nTF`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put `\prg_return_true:` and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code. Also the `\@@if_empty_if:o` is expanded once in `\tl_if_empty:oTF` for efficiency as well (and to reduce code doubling).

```

12796 \cs_new:Npn \__tl_if_empty_if:o #1
12797 {
12798     \if:w \scan_stop: \__kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
12799 }
12800 \exp_args:Nno \use:n
12801 { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
12802 {
12803     \__tl_if_empty_if:o {#1}
12804     \prg_return_true:
12805 \else:
12806     \prg_return_false:
12807 \fi:
12808 }

```

(End of definition for `\tl_if_empty:nTF` and `\__tl_if_empty_if:o`. This function is documented on page 115.)

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a *<token list>* is blank if and only if `\use_none:n <token list> ?` is empty after one expansion. The auxiliary `\__tl_if_empty_if:o` is a fast emptiness test, converting its argument to a string (after one expansion) and using the test `\if:w \scan_stop: ... \scan_stop:.`

```

\tl_if_blank:nTF
\tl_if_blank_p:V
\tl_if_blank_p:o
\tl_if_blank:nTF
\tl_if_blank:VTF
\tl_if_blank:oTF
\__tl_if_blank_p:NNw
12809 \exp_args:Nno \use:n
12810 { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } }
12811 {
12812   \__tl_if_empty_if:o { \use_none:n #1 ? }
12813   \prg_return_true:
12814   \else:
12815     \prg_return_false:
12816   \fi:
12817 }
12818 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
12819 { e , V , o } { p , T , F , TF }
```

(End of definition for `\tl_if_blank:nTF` and `\__tl_if_blank_p:NNw`. This function is documented on page 115.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` if and only if the two token list variables are equal.

```

\tl_if_eq_p:Nc
\tl_if_eq_p:cN
\tl_if_eq_p:cc
12820 \prg_new_eq_conditional:NNn \tl_if_eq:NN \cs_if_eq:NN { p , T , F , TF }
12821 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
12822 { Nc , c , cc } { p , TF , T , F }
```

`\tl_if_eq:NNTF` (End of definition for `\tl_if_eq:NNTF`. This function is documented on page 115.)

`\tl_if_eq:NcTF`

`\tl_if_eq:cNNTF`

`\tl_if_eq:ccTF`

Temporary storage.

(End of definition for `\l__tl_tmpa_tl` and `\l__tl_tmpb_tl`.)

`\tl_if_eq:NnTF` A simple store and compare routine.

(End of definition for `\tl_if_eq:NnTF`. This function is documented on page 115.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

12843     \exp_after:wN
12844 \group_end:
12845 \if_meaning:w \l__tl_tmpa_tl \l__tl_tmpb_tl
12846     \prg_return_true:
12847 \else:
12848     \prg_return_false:
12849 \fi:
12850 }
12851 \prg_generate_conditional_variant:Nnn \tl_if_eq:nn
12852 { nV , ne , nx , V , e , ee , x , xx }
12853 { TF , T , F }

```

(End of definition for `\tl_if_eq:nnTF`. This function is documented on page 115.)

`\tl_if_in:NnTF` See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable `\tl_if_in:NnTF` and pass it to `\tl_if_in:nnTF`.

```

\tl_if_in:NnTF 12854 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
\tl_if_in:NnTF 12855 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
\tl_if_in:NnTF 12856 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
\tl_if_in:NnTF 12857 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
12858 { NV , No , c , cV , co } { T , F , TF }

```

(End of definition for `\tl_if_in:NnTF`. This function is documented on page 116.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `\__tl_tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end, requires special care. To cater for this case, we insert `{ }{ }` between the two token lists. This marker may not appear in `#2` because of $\TeX$ limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nnTF` does not lead to unbalanced braces.

```

12859 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
12860 {
12861     \scan_stop:
12862     \if_false: { \fi:
12863         \cs_set:Npn \__tl_tmp:w ##1 #2 { }
12864         \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
12865         { \prg_return_false: } { \prg_return_true: }
12866     \if_false: } \fi:
12867 }
12868 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
12869 { V , VV , o , oo , nV , no } { T , F , TF }

```

(End of definition for `\tl_if_in:nnTF`. This function is documented on page 116.)

`\tl_if_novalue_p:n` Tests whether `##1` matches `-NoValue-` exactly (with suitable catcodes): this is similar to `\quark_if_nil:nTF`. The first argument of `\__tl_if_novalue:w` is empty if and only if `##1` starts with `-NoValue-`, while the second argument is empty if `##1` is exactly `-NoValue-` or if it has a question mark just following `-NoValue-`. In this second case,

however, the material after the first ?! remains and makes the emptiness test return false.

```

12870 \cs_set_protected:Npn \__tl_tmp:w #1
12871 {
12872   \prg_new_conditional:Npnn \tl_if_novalue:n ##1
12873   { p , T , F , TF }
12874   {
12875     \__tl_if_empty_if:o { \__tl_if_novalue:w {} ##1 {} ? ! #1 ? ? ! }
12876     \prg_return_true:
12877     \else:
12878     \prg_return_false:
12879     \fi:
12880   }
12881   \cs_new:Npn \__tl_if_novalue:w ##1 #1 ##2 ? ##3 ? ! { ##1 ##2 }
12882 }
12883 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End of definition for `\tl_if_novalue:nTF` and `\__tl_if_novalue:w`. This function is documented on page 116.)

<code>\tl_if_single_p:N</code> <code>\tl_if_single_p:c</code> <code>\tl_if_single:nTF</code> <code>\tl_if_single:cTF</code>	Expand the token list and feed it to <code>\tl_if_single:nTF</code> . <pre> 12884 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n } 12885 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT } 12886 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF } 12887 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF } 12888 \prg_generate_conditional_variant:Nnn \tl_if_single:N {c} { p , T , F , TF } </pre>

(End of definition for `\tl_if_single:nTF`. This function is documented on page 116.)

<code>\tl_if_single_p:n</code> <code>\tl_if_single:nTF</code> <code>__tl_if_single:nnw</code>	This test is similar to <code>\tl_if_empty:nTF</code> . Expanding <code>\use_none:nn #1 ??</code> once yields an empty result if #1 is blank, a single ? if #1 has a single item, and otherwise yields some tokens ending with ??. Then, <code>__kernel_tl_to_str:w</code> makes sure there are no odd category codes. An earlier version would compare the result to a single ? using string comparison, but the Lua call is slow in LuaTeX. Instead, <code>__tl_if_single:nnw</code> picks the second token in front of it. If #1 is empty, this token is the trailing ? and the <code>\if:w</code> test yields false. If #1 has a single item, the token is <code>\scan_stop:</code> and the <code>\if:w</code> test yields true. Otherwise, it is one of the characters resulting from <code>\tl_to_str:n</code> , and the <code>\if:w</code> test yields false. Note that <code>\if:w</code> and <code>__kernel_tl_to_str:w</code> are primitives that take care of expansion.

```

12889 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
12890 {
12891   \if:w \scan_stop: \exp_after:wN \__tl_if_single:nnw
12892     \__kernel_tl_to_str:w
12893     \exp_after:wN { \use_none:nn #1 ?? } \scan_stop: ? \s_tl_stop
12894     \prg_return_true:
12895   \else:
12896     \prg_return_false:
12897   \fi:
12898 }
12899 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \s_tl_stop {#2}

```

(End of definition for `\tl_if_single:nTF` and `\__tl_if_single:nnw`. This function is documented on page 116.)

`\tl_if_single_token_p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

`\tl_if_single_token:nTF`

```

12900 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
12901 {
12902   \tl_if_head_is_N_type:nTF {#1}
12903   { \_tl_if_empty_if:o { \use_none:n #1 } }
12904   {
12905     \tl_if_empty:nTF {#1}
12906     { \if_false: }
12907     { \_tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
12908   }
12909   \prg_return_true:
12910 \else:
12911   \prg_return_false:
12912 \fi:
12913 }

```

(End of definition for `\tl_if_single_token:nTF`. This function is documented on page 116.)

```

\tl_if_regex_match:nn
\tl_if_regex_match:Vn
\tl_if_regex_match:nN
\tl_if_regex_match:VN
12914 \prg_new_protected_conditional:Npnn \tl_if_regex_match:nn #1#2 { TF , T , F }
12915 {
12916   \regex_if_match:nnTF {#2} {#1}
12917   \prg_return_true: \prg_return_false:
12918 }
12919 \prg_generate_conditional_variant:Nnn \tl_if_regex_match:nn
12920 { V } { TF , T , F }
12921 \prg_new_protected_conditional:Npnn \tl_if_regex_match:nN #1#2 { TF , T , F }
12922 {
12923   \regex_if_match:NnTF #2 {#1}
12924   \prg_return_true: \prg_return_false:
12925 }
12926 \prg_generate_conditional_variant:Nnn \tl_if_regex_match:nN
12927 { V } { TF , T , F }

```

(End of definition for `\tl_if_regex_match:nn` and `\tl_if_regex_match:nN`. These functions are documented on page ??.)

57.8 Mapping over token lists

`\tl_map_function:nN` Expandable loop macro for token lists. We use the internal scan mark `\s__tl_stop` (defined later), which is not allowed to show up in the token list `#1` since it is internal to l3tl. This allows us a very fast test of whether some `<item>` is the end-marker `\s__tl_stop`, namely call `\_tl_use_none_delimit_by_s_stop:w <item> <function> \s__tl_stop`, which calls `<function>` if the `<item>` is the end-marker. To speed up the loop even more, only test one out of eight items, and once we hit one of the eight end-markers, go more slowly through the last few items of the list using `\_tl_map_function_end:w`.

`\tl_map_function:eN`

`\tl_map_function:NN`

`\tl_map_function:cN`

`\_tl_map_function:Nnnnnnnnn`

`\_tl_map_function_end:w`

`\_tl_use_none_delimit_by_s_stop:w`

```

12928 \cs_new:Npn \tl_map_function:nN #1#2
12929 {

```

```

12930     \__tl_map_function:Nnnnnnnnn #2 #1
12931     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12932     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12933     \prg_break_point:Nn \tl_map_break: { }
12934 }
12935 \cs_generate_variant:Nn \tl_map_function:nN { e }
12936 \cs_new:Npn \tl_map_function:NN
12937 { \exp_args:No \tl_map_function:nN }
12938 \cs_generate_variant:Nn \tl_map_function:NN { c }
12939 \cs_new:Npn \__tl_map_function:Nnnnnnnnn #1#2#3#4#5#6#7#8#9
12940 {
12941     \__tl_use_none_delimit_by_s_stop:w
12942     #9 \__tl_map_function_end:w \s__tl_stop
12943     #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
12944     \__tl_map_function:Nnnnnnnnn #1
12945 }
12946 \cs_new:Npn \__tl_map_function_end:w \s__tl_stop #1#2
12947 {
12948     \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12949     #1 {#2}
12950     \__tl_map_function_end:w \s__tl_stop
12951 }
12952 \cs_new:Npn \__tl_use_none_delimit_by_s_stop:w #1 \s__tl_stop { }

```

(End of definition for `\tl_map_function:nN` and others. These functions are documented on page 122.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter
`\tl_map_inline:Nn` `\g__kernel_prng_map_int` to make them nestable. We can also make use of `\__tl_-`
`\tl_map_inline:cn` `map_function:Nnnnnnnnn` from before.

```

12953 \cs_new_protected:Npn \tl_map_inline:nn #1#2
12954 {
12955     \int_gincr:N \g__kernel_prng_map_int
12956     \cs_gset_protected:cpn
12957     { \__tl_map_ \int_use:N \g__kernel_prng_map_int :w } ##1 {#2}
12958     \exp_args:Nc \__tl_map_function:Nnnnnnnnn
12959     { \__tl_map_ \int_use:N \g__kernel_prng_map_int :w }
12960     #1
12961     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12962     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12963     \prg_break_point:Nn \tl_map_break:
12964     { \int_gdecr:N \g__kernel_prng_map_int }
12965 }
12966 \cs_new_protected:Npn \tl_map_inline:Nn
12967 { \exp_args:No \tl_map_inline:nn }
12968 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End of definition for `\tl_map_inline:nn` and `\tl_map_inline:Nn`. These functions are documented on page 122.)

`\tl_map_tokens:nn` Much like the function mapping.

`\tl_map_tokens:Nn` `\cs_new:Npn \tl_map_tokens:nn #1#2`

`\tl_map_tokens:cn` `{`

`\__tl_map_tokens:nnnnnnnnn` `\__tl_map_tokens:nnnnnnnnn {#2} #1`

`\__tl_map_tokens_end:w` `\s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop`


```

12973     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12974     \prg_break_point:Nn \tl_map_break: { }
12975   }
12976   \cs_new:Npn \tl_map_tokens:Nn
12977     { \exp_args:No \tl_map_tokens:nn }
12978   \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
12979   \cs_new:Npn \__tl_map_tokens:nnnnnnnn #1#2#3#4#5#6#7#8#9
12980     {
12981       \__tl_use_none_delimit_by_s_stop:w
12982       #9 \__tl_map_tokens_end:w \s__tl_stop
12983       \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
12984       \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
12985       \__tl_map_tokens:nnnnnnnn {#1}
12986     }
12987   \cs_new:Npn \__tl_map_tokens_end:w \s__tl_stop \use:n #1#2
12988     {
12989       \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12990       #1 {#2}
12991       \__tl_map_tokens_end:w \s__tl_stop
12992     }

```

(End of definition for `\tl_map_tokens:nn` and others. These functions are documented on page 122.)

```

\tl_map_variable:nNn \tl_map_variable:nNn {(token list)} <tl var> {<action>} assigns <tl var> to each
\tl_map_variable:NNn element and executes <action>. The assignment to <tl var> is done after the quark test
\tl_map_variable:cNn so that this variable does not get set to a quark.
\__tl_map_variable:Nnn
12993 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
12994   { \tl_map_tokens:nn {#1} { \__tl_map_variable:Nnn #2 {#3} } }
12995 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
12996   { \tl_set:Nn #1 {#3} #2 }
12997 \cs_new_protected:Npn \tl_map_variable:NNn
12998   { \exp_args:No \tl_map_variable:nNn }
12999 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End of definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `\__tl_map_variable:Nnn`. These functions are documented on page 123.)

`\tl_map_break:` The break statements use the general `\prg_map_break:Nn`.
`\tl_map_break:n`

```

13000 \cs_new:Npn \tl_map_break:
13001   { \prg_map_break:Nn \tl_map_break: { } }
13002 \cs_new:Npn \tl_map_break:n
13003   { \prg_map_break:Nn \tl_map_break: }

```

(End of definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 123.)

57.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.
`\tl_to_str:o` 13004 \cs_generate_variant:Nn \tl_to_str:n { o , V , v , e }
`\tl_to_str:V`
`\tl_to_str:v` (End of definition for `\tl_to_str:n`. This function is documented on page 118.)
`\tl_to_str:e`

\tl_to_str:N These functions return the replacement text of a token list as a string.

\tl_to_str:c

```

13005 \cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
13006 \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End of definition for \tl_to_str:N. This function is documented on page 119.)

\tl_use:N Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.

\tl_use:c

```

13007 \cs_new:Npn \tl_use:N #1
13008 {
13009     \tl_if_exist:NTF #1 {#1}
13010     {
13011         \msg_expandable_error:nnn
13012             { kernel } { bad-variable } {#1}
13013     }
13014 }
13015 \cs_generate_variant:Nn \tl_use:N { c }

```

(End of definition for \tl_use:N. This function is documented on page 119.)

57.10 Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. __tl_count:n grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

\tl_count:V

\tl_count:v

\tl_count:e

\tl_count:o

\tl_count:N

\tl_count:c

__tl_count:n

```

13016 \cs_new:Npn \tl_count:n #1
13017 {
13018     \int_eval:n
13019         { 0 \tl_map_function:nN {#1} \__tl_count:n }
13020 }
13021 \cs_new:Npn \tl_count:N #1
13022 {
13023     \int_eval:n
13024         { 0 \tl_map_function:NN #1 \__tl_count:n }
13025 }
13026 \cs_new:Npn \__tl_count:n #1 { + 1 }
13027 \cs_generate_variant:Nn \tl_count:n { V , v , e , o }
13028 \cs_generate_variant:Nn \tl_count:N { c }

```

(End of definition for \tl_count:n, \tl_count:N, and __tl_count:n. These functions are documented on page 119.)

\tl_count_tokens:n The token count is computed through an \int_eval:n construction. Each 1+ is output to the left, into the integer expression, and the sum is ended by the \exp_end: inserted by __tl_act_end:wn (which is technically implemented as \c_zero_int). Somewhat a hack!

__tl_act_count_normal:nN

__tl_act_count_group:nn

__tl_act_count_space:n

```

13029 \cs_new:Npn \tl_count_tokens:n #1
13030 {
13031     \int_eval:n
13032     {
13033         \__tl_act:NNNn
13034         \__tl_act_count_normal:N

```

```

13035         \_tl_act_count_group:n
13036         \_tl_act_count_space:
13037         {#1}
13038     }
13039 }
13040 \cs_new:Npn \_tl_act_count_normal:N #1 { 1 + }
13041 \cs_new:Npn \_tl_act_count_space: { 1 + }
13042 \cs_new:Npn \_tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }

```

(End of definition for \tl_count_tokens:n and others. This function is documented on page 119.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \s__tl_stop.

```

13043 \cs_new:Npn \tl_reverse_items:n #1
13044 {
13045     \_tl_reverse_items:nwNwn #1 ?
13046     \s__tl_mark \_tl_reverse_items:nwNwn
13047     \s__tl_mark \_tl_reverse_items:wn
13048     \s__tl_stop { }
13049 }
13050 \cs_new:Npn \_tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
13051 {
13052     #3 #2
13053     \s__tl_mark \_tl_reverse_items:nwNwn
13054     \s__tl_mark \_tl_reverse_items:wn
13055     \s__tl_stop { {#1} #5 }
13056 }
13057 \cs_new:Npn \_tl_reverse_items:wn #1 \s__tl_stop #2
13058 { \_kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }

```

(End of definition for \tl_reverse_items:n, _tl_reverse_items:nwNwn, and _tl_reverse_items:wn. This function is documented on page 120.)

\tl_trim_spaces:n Trimming spaces from around the input is deferred to an internal function whose first argument is a *<continuation>*, which receives as a braced argument _tl_trim_mark: *<trimmed token list>*, and whose second argument is the token list to trim. The control sequence _tl_trim_mark: expands to nothing in a single expansion. In the case at hand, we take _kernel_exp_not:w \exp_after:wN as our continuation, so that space trimming behaves correctly within an e-type or x-type expansion.

```

13059 \cs_new:Npn \tl_trim_spaces:n
13060 { \_tl_trim_spaces:nn { \_kernel_exp_not:w \exp_after:wN } }
13061 \cs_new:Npn \tl_trim_left_spaces:n
13062 { \_tl_trim_left_spaces:nn { \_kernel_exp_not:w \exp_after:wN } }
13063 \cs_new:Npn \tl_trim_right_spaces:n
13064 { \_tl_trim_right_spaces:nn { \_kernel_exp_not:w \exp_after:wN } }
13065 \cs_generate_variant:Nn \tl_trim_spaces:n { V , v , e , o }
13066 \cs_generate_variant:Nn \tl_trim_left_spaces:n { V , v , e , o }
13067 \cs_generate_variant:Nn \tl_trim_right_spaces:n { V , v , e , o }
13068 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
13069 { \_tl_trim_spaces:nn { \exp_args:No #2 } { #1 } }
13070 \cs_new:Npn \tl_trim_left_spaces_apply:nN #1#2
13071 { \_tl_trim_left_spaces:nn { \exp_args:No #2 } { #1 } }
13072 \cs_new:Npn \tl_trim_right_spaces_apply:nN #1#2
13073 { \_tl_trim_right_spaces:nn { \exp_args:No #2 } { #1 } }

```

```

13074 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
13075 \cs_generate_variant:Nn \tl_trim_left_spaces_apply:nN { o }
13076 \cs_generate_variant:Nn \tl_trim_right_spaces_apply:nN { o }
13077 \cs_new_protected:Npn \tl_trim_spaces:N #1
13078 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
13079 \cs_new_protected:Npn \tl_trim_left_spaces:N #1
13080 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_left_spaces:n {#1} } }
13081 \cs_new_protected:Npn \tl_trim_right_spaces:N #1
13082 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_right_spaces:n {#1} } }
13083 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
13084 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
13085 \cs_new_protected:Npn \tl_gtrim_left_spaces:N #1
13086 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_left_spaces:n {#1} } }
13087 \cs_new_protected:Npn \tl_gtrim_right_spaces:N #1
13088 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_right_spaces:n {#1} } }
13089 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
13090 \cs_generate_variant:Nn \tl_trim_left_spaces:N { c }
13091 \cs_generate_variant:Nn \tl_trim_right_spaces:N { c }
13092 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
13093 \cs_generate_variant:Nn \tl_gtrim_left_spaces:N { c }
13094 \cs_generate_variant:Nn \tl_gtrim_right_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and `\__tl_trim_mark:`, and to get spaces at odd places in the definitions, we nest those in `\__tl_tmp:w`, which then receives a single space as its argument: `#1` is `␣`. Removing leading spaces is done with `\__tl_trim_spaces_auxi:w`, which loops until `\__tl_trim_mark:␣` matches the end of the token list: then `##1` is

```

\__tl_trim_mark:⟨left-trimmed token list⟩\s__tl_nil\__tl_trim_-
mark:\__tl_trim_spaces_auxi:w

```

and `##3` is `\__tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `\__tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `␣\s__tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `\__tl_trim_spaces_auxiv:w` puts the token list into a group, with a lingering `\__tl_trim_mark:` at the start (which will expand to nothing in one step of expansion), and feeds this to the *continuation*.

Trimming just leading spaces (see `\__tl_trim_left_spaces:nn`) also starts with `\__tl_trim_spaces_auxi:w`, but when it loops until `\__tl_trim_mark:␣` matches the end of the token list, `##1` is the same as in the `\__tl_trim_spaces:nn` case, *but* `##3` is `\__tl_trim_spaces_auxv:w`. Like `\__tl_trim_spaces_auxiv:w`, it hands the relevant tokens to the *continuation*. Trimming just trailing spaces (see `\__tl_trim_right_spaces:nn`) starts with the (second) loop `\__tl_trim_spaces_auxiii:w`.

```

\__tl_trim_spaces:nn
\__tl_trim_left_spaces:nn
\__tl_trim_right_spaces:nn
\__tl_trim_spaces_auxi:w
\__tl_trim_spaces_auxii:w
\__tl_trim_spaces_auxiii:w
\__tl_trim_spaces_auxiv:w
\__tl_trim_spaces_auxv:w
\__tl_trim_mark:

```

```

13095 \cs_set_protected:Npn \__tl_tmp:w #1
13096 {
13097   \cs_new:Npn \__tl_trim_spaces:nn ##1##2
13098   {
13099     \__tl_trim_spaces_auxi:w
13100     \__tl_trim_mark: ##2\s__tl_nil
13101     \__tl_trim_mark:\__tl_trim_spaces_auxi:w
13102     \__tl_trim_mark: #1
13103     \__tl_trim_mark:\__tl_trim_spaces_auxii:w
13104     {##1}

```

```

13105     }
13106 \cs_new:Npn \__tl_trim_left_spaces:nn ##1##2
13107 {
13108     \__tl_trim_spaces_auxi:w
13109     \__tl_trim_mark: ##2 \s__tl_nil
13110     \__tl_trim_mark: \__tl_trim_spaces_auxi:w
13111     \__tl_trim_mark: #1
13112     \__tl_trim_mark: \__tl_trim_spaces_auxv:w
13113     {##1}
13114 }
13115 \cs_new:Npn \__tl_trim_right_spaces:nn ##1##2
13116 {
13117     \__tl_trim_spaces_auxiii:w
13118     \__tl_trim_mark: ##2 \s__tl_nil \__tl_trim_spaces_auxiii:w
13119     #1 \s__tl_nil \__tl_trim_spaces_auxiv:w
13120     {##1}
13121 }
13122 \cs_new:Npn \__tl_trim_spaces_auxi:w
13123     ##1 \__tl_trim_mark: #1 ##2 \__tl_trim_mark: ##3
13124     { ##3 ##1 \__tl_trim_mark: ##2 \__tl_trim_mark: \__tl_trim_spaces_auxi:w }
13125 \cs_new:Npn \__tl_trim_spaces_auxii:w
13126     \__tl_trim_mark: ##1 \__tl_trim_mark: ##2 \__tl_trim_spaces_auxi:w
13127     \__tl_trim_mark: \__tl_trim_mark: \__tl_trim_spaces_auxi:w
13128     {
13129         \__tl_trim_spaces_auxiii:w
13130         \__tl_trim_mark: ##1 \__tl_trim_spaces_auxiii:w
13131         #1 \s__tl_nil \__tl_trim_spaces_auxiv:w
13132     }
13133 \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \s__tl_nil ##2
13134     { ##2 ##1 \s__tl_nil \__tl_trim_spaces_auxiii:w }
13135 \cs_new:Npn \__tl_trim_spaces_auxiv:w
13136     ##1 \s__tl_nil
13137     \__tl_trim_spaces_auxiii:w \s__tl_nil \__tl_trim_spaces_auxiii:w
13138     ##2
13139     { ##2 {##1} }
13140 \cs_new:Npn \__tl_trim_spaces_auxv:w
13141     ##1 \s__tl_nil
13142     \__tl_trim_mark: \__tl_trim_spaces_auxi:w \__tl_trim_mark:
13143     \__tl_trim_mark: \__tl_trim_spaces_auxi:w ##2
13144     { ##2 {##1} }
13145 \cs_new:Npn \__tl_trim_mark: {}
13146 }
13147 \__tl_tmp:w { ~ }

```

(End of definition for `\tl_trim_spaces:n` and others. These functions are documented on page 120.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn` (End of definition for `\tl_sort:Nn`, `\tl_gsort:Nn`, and `\tl_sort:nN`. These functions are documented on page 127.)

`\tl_gsort:Nn`

`\tl_gsort:cn`

`\tl_sort:nN`

57.11 The first token from a token list

`\tl_head:N` Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping over a list. The empty brace groups in `\tl_head:n`

`\tl_head:n`

`\tl_head:V`

`\tl_head:v`

`\tl_head:f`

`\tl_head:e`

`\__tl_head_auxi:nw`

`\__tl_head_auxii:n`

`\tl_head:w`

`\__tl_tl_head:w`

`\tl_tail:N`

`\tl_tail:n`

ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse its own code. More detail in <http://tex.stackexchange.com/a/70168>.

```

13148 \cs_new:Npn \tl_head:n #1
13149 {
13150   \__kernel_exp_not:w \tex_expanded:D
13151   { { \if_false: { \fi: \__tl_head_aux:n #1 { } } } }
13152 }
13153 \cs_new:Npn \__tl_head_aux:n #1
13154 {
13155   \__kernel_exp_not:w {#1}
13156   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
13157 }
13158 \cs_generate_variant:Nn \tl_head:n { V , v , f , e }
13159 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
13160 \cs_new:Npn \__tl_tl_head:w #1#2 \s__tl_stop {#1}
13161 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```

\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop

```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (i.e., that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimize the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

13162 \exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
13163 {
13164   \exp_after:wN \__kernel_exp_not:w
13165   \tl_if_blank:nTF {#1}
13166   { { } }
13167   { \exp_after:wN { \use_none:n #1 } }
13168 }
13169 \cs_generate_variant:Nn \tl_tail:n { V , v , f , e }
13170 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End of definition for `\tl_head:N` and others. These functions are documented on page 124.)

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```

\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
  \exp_not:N #2

```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two tokens: `^` and `\__tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and `#2`.

```

13171 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
13172 {
13173   \if_charcode:w
13174     \tl_if_head_is_N_type:nTF { #1 ? }
13175     { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
13176     { \str_head:n {#1} }
13177     \exp_not:N #2
13178     \prg_return_true:
13179   \else:
13180     \prg_return_false:
13181   \fi:
13182 }
13183 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
13184 { V , e , f } { p , TF , T , F }

```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return false.

```

13185 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
13186 {
13187   \if_catcode:w
13188     \tl_if_head_is_N_type:nTF { #1 ? }
13189     { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
13190     {
13191       \tl_if_head_is_group:nTF {#1}
13192       \c_group_begin_token
13193       \c_space_token
13194     }
13195     \exp_not:N #2
13196     \prg_return_true:
13197   \else:
13198     \prg_return_false:
13199   \fi:
13200 }
13201 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
13202 { V , e , o } { p , TF , T , F }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is `true`, and `\use_none:nnn` removes `#2` and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are

not nested because the arguments may contain unmatched primitive conditionals.

```

13203 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
13204 {
13205     \tl_if_head_is_N_type:nTF { #1 ? }
13206     \__tl_if_head_eq_meaning_normal:nN
13207     \__tl_if_head_eq_meaning_special:nN
13208     {#1} #2
13209 }
13210 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_meaning:nN
13211 { V , e } { p , TF , T , F }
13212 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
13213 {
13214     \exp_after:wN \if_meaning:w
13215     \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
13216     \prg_return_true:
13217 \else:
13218     \prg_return_false:
13219 \fi:
13220 }
13221 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
13222 {
13223     \if_charcode:w \str_head:n {#1} \exp_not:N #2
13224     \exp_after:wN \use_ii:nn
13225 \else:
13226     \prg_return_false:
13227 \fi:
13228     \use_none:n
13229     {
13230         \if_catcode:w \exp_not:N #2
13231             \tl_if_head_is_group:nTF {#1}
13232             { \c_group_begin_token }
13233             { \c_space_token }
13234         \prg_return_true:
13235     \else:
13236         \prg_return_false:
13237     \fi:
13238 }
13239 }

```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `\__tl_head_exp_not:w` does exactly that.

```

13240 \cs_new:Npn \__tl_head_exp_not:w #1 #2 \s__tl_stop
13241 { \exp_not:N #1 }

```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `\__tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return false.

```

13242 \cs_new:Npn \__tl_if_head_eq_empty_arg:w \exp_not:N #1
13243 { ? }

```

(End of definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 117.)

`\tl_if_head_is_N_type_p:n`
`\tl_if_head_is_N_type:nTF`
`\_tl_if_head_is_N_type_auxi:w`
`\_tl_if_head_is_N_type_auxii:n`

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, and when #1~ starts with {}, `\_tl_if_head_is_N_type_auxi:w` receives an empty argument hence produces `f` and removes everything before the first `\scan_stop:`. In the third case (except when #1~ starts with {}~), the second auxiliary removes the first copy of #1 that was used for the space test, then expands `\token_to_str:N` which hits the leading begin-group token, leaving a single closing brace to be compared with `\scan_stop:`. In the last case, `\token_to_str:N` does not change the brace balance so that only `\scan_stop: \scan_stop:` remain, making the character code test true. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the true branch of the conditional.

```

13244 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
13245 {
13246   \if:w
13247     \if_false: { \fi: \_tl_if_head_is_N_type_auxi:w #1 ~ }
13248     { \exp_after:wN { \token_to_str:N #1 } }
13249     \scan_stop: \scan_stop:
13250     \prg_return_true:
13251   \else:
13252     \prg_return_false:
13253   \fi:
13254 }
13255 \exp_args:Nno \use:n { \cs_new:Npn \_tl_if_head_is_N_type_auxi:w #1 ~ }
13256 {
13257   \tl_if_empty:nTF {#1}
13258   { f \exp_after:wN \use_none:nn }
13259   { \exp_after:wN \_tl_if_head_is_N_type_auxii:n }
13260   \exp_after:wN { \if_false: } \fi:
13261 }
13262 \cs_new:Npn \_tl_if_head_is_N_type_auxii:n #1
13263 { \exp_after:wN \use_none:n \exp_after:wN }

```

(End of definition for `\tl_if_head_is_N_type:nTF`, `\_tl_if_head_is_N_type_auxi:w`, and `\_tl_if_head_is_N_type_auxii:n`. This function is documented on page 118.)

`\tl_if_head_is_group_p:n`
`\tl_if_head_is_group:nTF`
`\_tl_if_head_is_group_fi_false:w`

Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance. The extra ? caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```

13264 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
13265 {
13266   \if:w
13267     \exp_after:wN \use_none:n
13268     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
13269     \scan_stop: \scan_stop:
13270     \_tl_if_head_is_group_fi_false:w
13271   \fi:
13272   \if_true:
13273     \prg_return_true:
13274   \else:
13275     \prg_return_false:
13276   \fi:

```

```

13277 }
13278 \cs_new:Npn \__tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }

```

(End of definition for `\tl_if_head_is_group:nTF` and `\__tl_if_head_is_group_fi_false:w`. This function is documented on page 117.)

`\tl_if_head_is_space_p:n` The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`.
`\tl_if_head_is_space:nTF` If that is a single `\prg_do_nothing:` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from $\TeX$ in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

13279 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
13280 {
13281   \if:w
13282     \if_false: { \fi: \__tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
13283     \scan_stop: \scan_stop:
13284     \prg_return_true:
13285   \else:
13286     \prg_return_false:
13287   \fi:
13288 }
13289 \exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
13290 {
13291   \__tl_if_empty_if:o {#1} \else: f \fi:
13292   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
13293 }

```

(End of definition for `\tl_if_head_is_space:nTF` and `\__tl_if_head_is_space:w`. This function is documented on page 118.)

57.12 Token by token changes

`\s__tl_act_stop` The `\__tl_act...` functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only `\s__tl_act_stop` may not appear in the token lists manipulated by `\__tl_act:NNNn` functions.

```

13294 \scan_new:N \s__tl_act_stop

```

(End of definition for `\s__tl_act_stop`.)

`\__tl_act:NNNn` To help control the expansion, `\__tl_act:NNNn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. This way no internal token of it can be accidentally end up in the input stream. Because `\s__tl_act_stop` can't appear without braces around it in the argument `#1` of `\__tl_act_loop:w`, we can use this marker to set up a fast test for leading spaces.

```

13295 \cs_set_protected:Npn \__tl_tmp:w #1
13296 {
13297   \cs_new:Npn \__tl_act_if_head_is_space:nTF ##1
13298   {
13299     \__tl_act_if_head_is_space:w
13300     \s__tl_act_stop ##1 \s__tl_act_stop \__tl_act_if_head_is_space_true:w
13301     \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn

```

```

13302     }
13303     \cs_new:Npn \__tl_act_if_head_is_space:w
13304         ##1 \s__tl_act_stop #1 ##2 \s__tl_act_stop
13305     {}
13306     \cs_new:Npn \__tl_act_if_head_is_space_true:w
13307         \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn ##1 ##2
13308         {##1}
13309     }
13310     \__tl_tmp:w { ~ }

```

(We expand the definition `\__tl_act_if_head_is_space:nTF` when setting up `\__tl_act_loop:w`, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the “group” case, we may have reached `\s__tl_act_stop`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that `\__tl_act_space:wwNNN` gobbles the space.

```

13311 \exp_args:Nne \use:n { \cs_new:Npn \__tl_act_loop:w #1 \s__tl_act_stop }
13312 {
13313     \exp_not:o { \__tl_act_if_head_is_space:nTF {#1} }
13314     \exp_not:N \__tl_act_space:wwNNN
13315     {
13316         \exp_not:o { \tl_if_head_is_group:nTF {#1} }
13317         \exp_not:N \__tl_act_group:nwNNN
13318         \exp_not:N \__tl_act_normal:NwNNN
13319     }
13320     \exp_not:n {#1} \s__tl_act_stop
13321 }
13322 \cs_undefine:N \__tl_act_if_head_is_space:nTF
13323 \cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
13324 {
13325     #3 #1
13326     \__tl_act_loop:w #2 \s__tl_act_stop
13327     #3
13328 }
13329 \cs_new:Npn \__tl_use_none_delimit_by_s_act_stop:w #1 \s__tl_act_stop { }
13330 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
13331 { \group_align_safe_end: \exp_end: #2 }
13332 \cs_new:Npn \__tl_act_group:nwNNN #1 #2 \s__tl_act_stop #3#4#5
13333 {
13334     \__tl_use_none_delimit_by_s_act_stop:w #1 \__tl_act_end:wn \s__tl_act_stop
13335     #5 {#1}
13336     \__tl_act_loop:w #2 \s__tl_act_stop
13337     #3 #4 #5
13338 }
13339 \exp_last_unbraced:NNo
13340 \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \s__tl_act_stop #2#3
13341 {
13342     #3
13343     \__tl_act_loop:w #1 \s__tl_act_stop
13344     #2 #3
13345 }

```

`\__tl_act:NNNn` loops over tokens, groups, and spaces in #4. `{\s_@@_act_stop}` serves

as the end of token list marker, the ? after it avoids losing outer braces. The result is stored as an argument for the dummy function `\__tl_act_result:n`.

```

13346 \cs_new:Npn \__tl_act:NNNn #1#2#3#4
13347 {
13348   \group_align_safe_begin:
13349   \__tl_act_loop:w #4 { \s__tl_act_stop } ? \s__tl_act_stop
13350   #1 #3 #2
13351   \__tl_act_result:n { }
13352 }

```

Typically, the output is done to the right of what was already output, using `\__tl_act_output:n`, but for the `\__tl_act_reverse` functions, it should be done to the left.

```

13353 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
13354 { #2 \__tl_act_result:n { #3 #1 } }
13355 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
13356 { #2 \__tl_act_result:n { #1 #3 } }

```

(End of definition for `\__tl_act:NNNn` and others.)

`\tl_reverse:n` The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `\__tl_act:NNNn`. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group.

```

\tl_reverse:o
\tl_reverse:V
\tl_reverse:f
\tl_reverse:e
\__tl_reverse_normal:nN
\__tl_reverse_group_preserve:nn
\__tl_reverse_space:n
13357 \cs_new:Npn \tl_reverse:n #1
13358 {
13359   \__kernel_exp_not:w \exp_after:wN
13360   {
13361     \exp:w
13362     \__tl_act:NNNn
13363     \__tl_reverse_normal:N
13364     \__tl_reverse_group_preserve:n
13365     \__tl_reverse_space:
13366     {#1}
13367   }
13368 }
13369 \cs_generate_variant:Nn \tl_reverse:n { o , V , f , e }
13370 \cs_new:Npn \__tl_reverse_normal:N
13371 { \__tl_act_reverse_output:n }
13372 \cs_new:Npn \__tl_reverse_group_preserve:n #1
13373 { \__tl_act_reverse_output:n { {#1} } }
13374 \cs_new:Npn \__tl_reverse_space:
13375 { \__tl_act_reverse_output:n { ~ } }

```

(End of definition for `\tl_reverse:n` and others. This function is documented on page 119.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```

\tl_reverse:c
\tl_greverse:N
\tl_greverse:c
13376 \cs_new_protected:Npn \tl_reverse:N #1
13377 { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
13378 \cs_new_protected:Npn \tl_greverse:N #1
13379 { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
13380 \cs_generate_variant:Nn \tl_reverse:N { c }
13381 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End of definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 120.)

57.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\__tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

`\tl_item:Nn`

`\tl_item:cn`

```

\__tl_item_aux:nn 13382 \cs_new:Npn \tl_item:nn #1#2
\__tl_item:nn     13383 {
                  13384   \exp_args:Nf \__tl_item:nn
                  13385   { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
                  13386   #1
                  13387   \q__tl_recursion_tail
                  13388   \prg_break_point:
                  13389 }
13390 \cs_new:Npn \__tl_item_aux:nn #1#2
13391 {
13392   \int_compare:nNnTF {#1} < 0
13393   { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
13394   {#1}
13395 }
13396 \cs_new:Npn \__tl_item:nn #1#2
13397 {
13398   \__tl_if_recursion_tail_break:nN {#2} \prg_break:
13399   \int_compare:nNnTF {#1} = 1
13400   { \prg_break:n { \exp_not:n {#2} } }
13401   { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
13402 }
13403 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
13404 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End of definition for `\tl_item:nn` and others. These functions are documented on page 125.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

`\tl_rand_item:N`

`\tl_rand_item:c`

```

13405 \cs_new:Npn \tl_rand_item:n #1
13406 {
13407   \tl_if_blank:nF {#1}
13408   { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
13409 }
13410 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
13411 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End of definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 125.)

`\__kernel_int_sep:` See comments in the kernel functions file: has to be set up here as this is the first module where it is needed.

```
13412 \cs_new_eq:NN \__kernel_int_sep: \tex_let:D
```

(End of definition for `\__kernel_int_sep:`.)

`\__tl_sep:`

```
13413 \cs_new_eq:NN \__tl_sep: \__kernel_int_sep:
```

(End of definition for `\__tl_sep:`.)

\tl_range:Nnn

\tl_range:cnn

\tl_range:nnn

__tl_range:Nnnn

__tl_range:nnnNn

__tl_range:nnNn

__tl_range_skip:w

__tl_range:w

__tl_range_skip_spaces:n

__tl_range_collect:nn

__tl_range_collect:ff

__tl_range_collect_space:nw

__tl_range_collect_N:nN

__tl_range_collect_group:nN

To avoid checking for the end of the token list at every step, start by counting the number l of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, `\__tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either `\__tl_range:w` or the token list itself. If nothing should be kept, leave `{}`: this stops the `f`-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `\__tl_range_skip:w` to delete #1 items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `\__tl_range_braced:w` sets up `\__tl_range_collect_braced:w` which stores items one by one in an argument after the `\__tl_sep:.` Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```
13414 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
13415 \cs_generate_variant:Nn \tl_range:Nnn { c }
13416 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
13417 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
13418 {
13419   \tl_head:f
13420   {
13421     \exp_args:Nf \__tl_range:nnnNn
13422     { \tl_count:n {#2} } {#3} {#4} #1 {#2}
13423   }
13424 }
13425 \cs_new:Npn \__tl_range:nnnNn #1#2#3
13426 {
13427   \exp_args:Nff \__tl_range:nnNn
13428   {
13429     \exp_args:Nf \__tl_range_normalize:nn
13430     { \int_eval:n { #2 - 1 } } {#1}
13431   }
13432   {
13433     \exp_args:Nf \__tl_range_normalize:nn
13434     { \int_eval:n {#3} } {#1}
13435   }
13436 }
13437 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
13438 {
13439   \if_int_compare:w #2 > #1 \exp_stop_f: \else:
13440     \exp_after:wN { \exp_after:wN }
13441   \fi:
13442   \exp_after:wN #3
13443   \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN \__tl_sep:
13444   \exp_after:wN { \exp:w \__tl_range_skip:w #1 \__tl_sep: { } #4 }
13445 }
13446 \cs_new:Npn \__tl_range_skip:w #1 \__tl_sep: #2
13447 {
13448   \if_int_compare:w #1 > \c_zero_int
13449     \exp_after:wN \__tl_range_skip:w
13450     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN \__tl_sep:
13451   \else:
13452     \exp_after:wN \exp_end:
13453   \fi:
```

```

13454 }
13455 \cs_new:Npn \__tl_range:w #1 \__tl_sep: #2
13456 {
13457   \exp_args:Nf \__tl_range_collect:nn
13458   { \__tl_range_skip_spaces:n {#2} } {#1}
13459 }
13460 \cs_new:Npn \__tl_range_skip_spaces:n #1
13461 {
13462   \tl_if_head_is_space:nTF {#1}
13463   { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
13464   { { } #1 }
13465 }
13466 \cs_new:Npn \__tl_range_collect:nn #1#2
13467 {
13468   \int_compare:nNnTF {#2} = 0
13469   {#1}
13470   {
13471     \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
13472     {
13473       \exp_args:Nf \__tl_range_collect:nn
13474       { \__tl_range_collect_space:nw #1 }
13475       {#2}
13476     }
13477     {
13478       \__tl_range_collect:ff
13479       {
13480         \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
13481         { \__tl_range_collect_N:nN }
13482         { \__tl_range_collect_group:nn }
13483         #1
13484       }
13485       { \int_eval:n { #2 - 1 } }
13486     }
13487   }
13488 }
13489 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
13490 \cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
13491 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
13492 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End of definition for `\tl_range:Nnn` and others. These functions are documented on page 126.)

`\__tl_range_normalize:nn` This function converts an *<index>* argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the *<index>* #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

13493 \cs_new:Npn \__tl_range_normalize:nn #1#2
13494 {
13495   \int_eval:n
13496   {
13497     \if_int_compare:w #1 < \c_zero_int
13498     \if_int_compare:w #1 < -#2 \exp_stop_f:
13499     0
13500     \else:

```

```

13501         #1 + #2 + 1
13502     \fi:
13503 \else:
13504     \if_int_compare:w #1 < #2 \exp_stop_f:
13505         #1
13506     \else:
13507         #2
13508     \fi:
13509 \fi:
13510 }
13511 }

```

(End of definition for `\__tl_range_normalize:nn`.)

57.14 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `\tl_show:c` `\__kernel_register_show:N`).

`\tl_log:N` `\tl_log:c` `\__tl_show:NN`

```

13512 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
13513 \cs_generate_variant:Nn \tl_show:N { c }
13514 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
13515 \cs_generate_variant:Nn \tl_log:N { c }
13516 \cs_new_protected:Npn \__tl_show:NN #1#2
13517 {
13518     \__kernel_chk_defined:NT #2
13519     {
13520         \exp_args:Nf \tl_if_empty:nTF
13521             { \cs_prefix_spec:N #2 \cs_parameter_spec:N #2 }
13522             {
13523                 \exp_args:Ne #1
13524                 { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
13525             }
13526         {
13527             \msg_error:nneee { kernel } { bad-type }
13528             { \token_to_str:N #2 } { \token_to_meaning:N #2 } { tl }
13529         }
13530     }
13531 }

```

(End of definition for `\tl_show:N`, `\tl_log:N`, and `\__tl_show:NN`. These functions are documented on page 121.)

`\tl_show:n` Many show functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

`\__tl_show:w` The token list `\l__tl_tmpa_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `\__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by T_EX, and that `\errorcontextlines` is `-1` to avoid printing irrelevant context.

```

13532 \cs_new_protected:Npn \tl_show:n #1

```



```

13533 { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
13534 \cs_generate_variant:Nn \tl_show:n { e , x }
13535 \cs_new_protected:Npn \__tl_show:n #1
13536 {
13537   \tl_set:Nf \l__tl_tmpa_tl { \__tl_show:w #1 \s__tl_stop }
13538   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
13539   {
13540     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
13541     {
13542       \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
13543       { \exp_after:wN \l__tl_tmpa_tl }
13544     }
13545   }
13546 }
13547 \cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}

```

(End of definition for `\tl_show:n`, `\__tl_show:n`, and `\__tl_show:w`. This function is documented on page 121.)

`\tl_log:n` Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

\tl_log:e
\tl_log:x
13548 \cs_new_protected:Npn \tl_log:n #1
13549 { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
13550 \cs_generate_variant:Nn \tl_log:n { e , x }

```

(End of definition for `\tl_log:n`. This function is documented on page 121.)

`\__kernel_chk_tl_type:NnnT` Helper for checking that `#1` has the correct internal structure to be of a certain type. Make sure that it is defined and that it is a token list, namely a macro with no `\long` nor `\protected` prefix. Then compare `#1` to an attempt at reconstructing a valid structure of the given type using `#2` (see implementation of `\seq_show:N` for instance). If that is successful run the requested code `#4`.

```

13551 \cs_new_protected:Npn \__kernel_chk_tl_type:NnnT #1#2#3#4
13552 {
13553   \__kernel_chk_defined:NT #1
13554   {
13555     \exp_args:Nf \tl_if_empty:nTF
13556     { \cs_prefix_spec:N #1 \cs_parameter_spec:N #1 }
13557     {
13558       \tl_set:Ne \l__tl_tmpa_tl {#3}
13559       \tl_if_eq:NNTF #1 \l__tl_tmpa_tl
13560       {#4}
13561       {
13562         \msg_error:nneeee { kernel } { bad-type }
13563         { \token_to_str:N #1 } { \tl_to_str:N #1 }
13564         {#2} { \tl_to_str:N \l__tl_tmpa_tl }
13565       }
13566     }
13567     {
13568       \msg_error:nneee { kernel } { bad-type }
13569       { \token_to_str:N #1 } { \token_to_meaning:N #1 } {#2}
13570     }
13571   }
13572 }

```

(End of definition for `\__kernel_chk_tl_type:NnnT`.)

57.15 Internal scan marks

`\s__tl_nil` Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some `!3tl` functions.

`\s__tl_mark`

`\s__tl_stop`

```
13573 \scan_new:N \s__tl_nil
13574 \scan_new:N \s__tl_mark
13575 \scan_new:N \s__tl_stop
```

(End of definition for \s__tl_nil, \s__tl_mark, and \s__tl_stop.)

57.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

`\g_tmpb_tl`

```
13576 \tl_new:N \g_tmpa_tl
13577 \tl_new:N \g_tmpb_tl
```

(End of definition for \g_tmpa_tl and \g_tmpb_tl. These variables are documented on page 131.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

`\l_tmpb_tl`

```
13578 \tl_new:N \l_tmpa_tl
13579 \tl_new:N \l_tmpb_tl
```

(End of definition for \l_tmpa_tl and \l_tmpb_tl. These variables are documented on page 131.)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```
13580 \cs_undefine:N \__tl_tmp:w
13581 </code>
```

Chapter 58

l3tl-build implementation

```
13582 <*code>
```

```
13583 <@@=tl>
```

Between `\tl_build_begin:N <tl var>` and `\tl_build_end:N <tl var>`, the `<tl var>` has the structure

```
\exp_end: ... \exp_end: \_tl_build_last:NNn <assignment> <next tl>
{<left>} <right>
```

where `<right>` is not braced. The “data” it represents is `<left>` followed by the “data” of `<next tl>` followed by `<right>`. The `<next tl>` is a token list variable whose name is that of `<tl var>` followed by `'`. There are between 0 and 4 `\exp_end:` to keep track of when `<left>` and `<right>` should be put into the `<next tl>`. The `<assignment>` is `\cs_set_nopar:Npe` if the variable is local, and `\cs_gset_nopar:Npe` if it is global.

```
\tl_build_begin:N
\tl_build_gbegin:N
\_tl_build_begin:NN
\_tl_build_begin:NNN
```

First construct the `<next tl>`: using a prime here conflicts with the usual `expl3` convention but we need a name that can be derived from `#1` without any external data such as a counter. Empty that `<next tl>` and setup the structure. The local and global versions only differ by a single function `\cs_(g)set_nopar:Npe` used for all assignments: this is important because only that function is stored in the `<tl var>` and `<next tl>` for subsequent assignments. In principle `\_tl_build_begin:NNN` could use `\tl_(g)clear_new:N` to empty `#1` and make sure it is defined, but logging the definition does not seem useful so we just do `#3 #1 {}` to clear it locally or globally as appropriate.

```
13584 \cs_new_protected:Npn \tl_build_begin:N #1
13585 { \_tl_build_begin:NN \cs_set_nopar:Npe #1 }
13586 \cs_new_protected:Npn \tl_build_gbegin:N #1
13587 { \_tl_build_begin:NN \cs_gset_nopar:Npe #1 }
13588 \cs_new_protected:Npn \_tl_build_begin:NN #1#2
13589 { \exp_args:Nc \_tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
13590 \cs_new_protected:Npn \_tl_build_begin:NNN #1#2#3
13591 {
13592   #3 #1 { }
13593   #3 #2
13594   {
13595     \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
13596     \exp_not:n { \_tl_build_last:NNn #3 #1 { } }
13597   }
13598 }
```

(End of definition for `\tl_build_begin:N` and others. These functions are documented on page 132.)

`\tl_build_put_right:Nn` Similar to `\tl_put_right:Nn`, but apply `\exp:w` to #1. Most of the time this just removes one `\exp_end:`. When there are none left, `\__tl_build_last:NNn` is expanded instead.

`\tl_build_put_right:Ne` It resets the definition of the `\tl var` by ending the `\exp_not:n` and the definition

`\tl_build_put_right:Nx` early. Then it makes sure the `\next tl` (its argument #1) is set-up and starts a new

`\tl_build_gput_right:Nn` definition. Then `\__tl_build_put:nn` and `\__tl_build_put:nw` place the `\left` part

`\tl_build_gput_right:Ne` of the original `\tl var` as appropriate for the definition of the `\next tl` (the `\right`)

`\tl_build_gput_right:Nx` part is left in the right place without ever becoming a macro argument). We use `\exp_`

`\__tl_build_last:NNn` `after:wN` rather than some `\exp_args:No` to avoid reading arguments that are likely

`\__tl_build_put:nn` very long token lists. We use `\cs_(g)set_nopar:Npe` rather than `\tl_(g)set:Ne` partly

`\__tl_build_put:nw` for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an e-expansion of the second argument.

```

13599 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
13600 {
13601   \cs_set_nopar:Npe #1
13602   { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
13603 }
13604 \cs_generate_variant:Nn \tl_build_put_right:Nn { Ne , Nx }
13605 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
13606 {
13607   \cs_gset_nopar:Npe #1
13608   { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
13609 }
13610 \cs_generate_variant:Nn \tl_build_gput_right:Nn { Ne , Nx }
13611 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
13612 {
13613   \if_false: { { \fi:
13614     \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
13615     \__tl_build_last:NNn #1 #2 { }
13616   }
13617 }
13618 \if_meaning:w \c_empty_tl #2
13619   \__tl_build_begin:NN #1 #2
13620 \fi:
13621 #1 #2
13622 {
13623   \__kernel_exp_not:w \exp_after:wN
13624   {
13625     \exp:w \if_false: } } \fi:
13626     \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
13627 }
13628 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
13629 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
13630 { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }

```

(End of definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 132.)

`\tl_build_put_left:Nn` See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_`

`\tl_build_put_left:Ne` `build_put_left_right:NNn`, by just adding the `\right` material after the `{\left}` in

`\tl_build_put_left:Nx` the e-expanding assignment.

`\tl_build_gput_left:Nn`

`\tl_build_gput_left:Ne`

`\tl_build_gput_left:Nx`

`\__tl_build_put_left:NNn`

```

13631 \cs_new_protected:Npn \tl_build_put_left:Nn #1
13632 { \__tl_build_put_left:NNn \cs_set_nopar:Npe #1 }
13633 \cs_generate_variant:Nn \tl_build_put_left:Nn { Ne , Nx }
13634 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
13635 { \__tl_build_put_left:NNn \cs_gset_nopar:Npe #1 }
13636 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Ne , Nx }
13637 \cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
13638 {
13639     #1 #2
13640     {
13641         \__kernel_exp_not:w \exp_after:wN
13642         {
13643             \exp:w \exp_after:wN \__tl_build_put:nn
13644             \exp_after:wN {#2} {#3}
13645         }
13646     }
13647 }

```

(End of definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `\__tl_build_put_left:NNn`. These functions are documented on page 132.)

`\tl_build_end:N` Get the data then clear the `<next tl>` recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_str:N`. The local/global scope is checked by `\tl_set:Ne` or `\tl_gset:Ne`.

`\tl_build_gend:N`

`\__tl_build_end_loop:NN`

```

13648 \cs_new_protected:Npn \tl_build_end:N #1
13649 {
13650     \__tl_build_get:NNN \__kernel_tl_set:Nx #1 #1
13651     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
13652 }
13653 \cs_new_protected:Npn \tl_build_gend:N #1
13654 {
13655     \__tl_build_get:NNN \__kernel_tl_gset:Nx #1 #1
13656     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
13657 }
13658 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
13659 {
13660     \if_meaning:w \c_empty_tl #1
13661     \exp_after:wN \use_none:nnnnnn
13662     \fi:
13663     #2 #1
13664     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
13665 }

```

(End of definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `\__tl_build_end_loop:NN`. These functions are documented on page 133.)

`\tl_build_get_intermediate:NN`

```

13666 \cs_new_protected:Npn \tl_build_get_intermediate:NN
13667 { \__tl_build_get:NNN \__kernel_tl_set:Nx }

```

(End of definition for `\tl_build_get_intermediate:NN`. This function is documented on page 133.)

`\__tl_build_get:NNN` The idea is to expand the `<tl var>` then the `<next tl>` and so on, all within an expanding assignment, and wrap as appropriate in `\exp_not:n`. The various `<left>` parts are left in the assignment as we go, which enables us to expand the `<next tl>` at

`\__tl_build_get:w`

`\__tl_build_get_end:w`

the right place. The various *⟨right⟩* parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the *⟨right⟩* parts together.

```

13668 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
13669   { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
13670 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
13671   {
13672     \exp_not:n {#4}
13673     \if_meaning:w \c_empty_tl #3
13674       \exp_after:wN \__tl_build_get_end:w
13675     \fi:
13676     \exp_after:wN \__tl_build_get:w #3
13677   }
13678 \cs_new:Npn \__tl_build_get_end:w #1#2#3
13679   { \__kernel_exp_not:w \exp_after:wN { \if_false: } \fi: }

```

(End of definition for `\__tl_build_get:NNN`, `\__tl_build_get:w`, and `\__tl_build_get_end:w`.)

```

13680 ⟨/code⟩

```

Chapter 59

l3str implementation

```
13681 ⟨*code⟩
13682 ⟨@@=str⟩
```

59.1 Internal auxiliaries

`\s__str_mark` Internal scan marks.

```
\s__str_stop 13683 \scan_new:N \s__str_mark
13684 \scan_new:N \s__str_stop
```

(End of definition for \s__str_mark and \s__str_stop.)

`\_str_use_none_delimit_by_s_stop:w` Functions to gobble up to a scan mark.

```
\_str_use_i_delimit_by_s_stop:nw 13685 \cs_new:Npn \_str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
13686 \cs_new:Npn \_str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
```

(End of definition for _str_use_none_delimit_by_s_stop:w and _str_use_i_delimit_by_s_stop:nw.)

`\q__str_recursion_tail` Internal recursion quarks.

```
\q__str_recursion_stop 13687 \quark_new:N \q__str_recursion_tail
13688 \quark_new:N \q__str_recursion_stop
```

(End of definition for \q__str_recursion_tail and \q__str_recursion_stop.)

`\_str_if_recursion_tail_break:NN` Functions to query recursion quarks.

```
\_str_if_recursion_tail_stop_do:Nn 13689 \__kernel_quark_new_test:N \_str_if_recursion_tail_break:NN
13690 \__kernel_quark_new_test:N \_str_if_recursion_tail_stop_do:Nn
```

(End of definition for _str_if_recursion_tail_break:NN and _str_if_recursion_tail_stop_do:Nn.)

59.2 Creating and setting string variables

`\str_new:N` A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```

\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc
13691 \group_begin:
13692   \cs_set_protected:Npn \__str_tmp:n #1
13693   {
13694     \tl_if_blank:nF {#1}
13695     {
13696       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
13697       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
13698       \__str_tmp:n
13699     }
13700   }
13701   \__str_tmp:n
13702   { new }
13703   { use }
13704   { clear }
13705   { gclear }
13706   { clear_new }
13707   { gclear_new }
13708   { }
13709 \group_end:
13710 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
13711 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
13712 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
13713 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
13714 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
13715 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
13716 \cs_generate_variant:Nn \str_concat:NNN { ccc }
13717 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }

```

(End of definition for `\str_new:N` and others. These functions are documented on page 135.)

`\str_set:Nn` Similar to corresponding `l3tl` base functions, except that `\__kernel_exp_not:w` is replaced with `\__kernel_tl_to_str:w`. Just like token list, string constants use `\cs_gset_nopar:Npe` instead of `\__kernel_tl_gset:Nx` so that the scope checking for `c` is applied when `l3debug` is used. To maintain backward compatibility, in `\str_(g)put_left:Nn` and `\str_(g)put_right:Nn`, contents of string variables are wrapped in `\__kernel_exp_not:w` to prevent further expansion.

```

13718 \cs_new_protected:Npn \str_set:Nn #1#2
13719 { \__kernel_tl_set:Nx #1 { \__kernel_tl_to_str:w {#2} } }
13720 \cs_gset_protected:Npn \str_gset:Nn #1#2
13721 { \__kernel_tl_gset:Nx #1 { \__kernel_tl_to_str:w {#2} } }
13722 \cs_new_protected:Npn \str_const:Nn #1#2
13723 {
13724   \__kernel_chk_if_free_cs:N #1
13725   \cs_gset_nopar:Npe #1 { \__kernel_tl_to_str:w {#2} }
13726 }
13727 \cs_new_protected:Npn \str_put_left:Nn #1#2
13728 {
13729   \__kernel_tl_set:Nx #1
13730   { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
13731 }
\str_set:NV
\str_set:Ne
\str_set:Nx
\str_set:cn
\str_set:cV
\str_set:ce
\str_set:cx
\str_gset:Nn
\str_gset:NV
\str_gset:Ne
\str_gset:Nx
\str_gset:cn
\str_gset:cV
\str_gset:ce
\str_gset:cx
\str_const:Nn
\str_const:NV
\str_const:Ne
\str_const:Nx
\str_const:cn
\str_const:cV
\str_const:ce
\str_const:cx
\str_put_left:Nn
\str_put_left:NV
\str_put_left:Ne
\str_put_left:Nx
\str_put_left:cn

```



```

13732 \cs_new_protected:Npn \str_gput_left:Nn #1#2
13733 {
13734   \__kernel_tl_gset:Nx #1
13735   { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
13736 }
13737 \cs_new_protected:Npn \str_put_right:Nn #1#2
13738 {
13739   \__kernel_tl_set:Nx #1
13740   { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
13741 }
13742 \cs_new_protected:Npn \str_gput_right:Nn #1#2
13743 {
13744   \__kernel_tl_gset:Nx #1
13745   { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
13746 }
13747 \cs_generate_variant:Nn \str_set:Nn { NV , Ne , Nx , c , cV , ce , cx }
13748 \cs_generate_variant:Nn \str_gset:Nn { NV , Ne , Nx , c , cV , ce , cx }
13749 \cs_generate_variant:Nn \str_const:Nn { NV , Ne , Nx , c , cV , ce , cx }
13750 \cs_generate_variant:Nn \str_put_left:Nn { NV , Ne , Nx , c , cV , ce , cx }
13751 \cs_generate_variant:Nn \str_gput_left:Nn { NV , Ne , Nx , c , cV , ce , cx }
13752 \cs_generate_variant:Nn \str_put_right:Nn { NV , Ne , Nx , c , cV , ce , cx }
13753 \cs_generate_variant:Nn \str_gput_right:Nn { NV , Ne , Nx , c , cV , ce , cx }

```

(End of definition for `\str_set:Nn` and others. These functions are documented on page 135.)

59.3 Modifying string variables

`\str_replace_all:Nnn` Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then

`\str_replace_all:cnn` the code is a much simplified version of the token list code because neither the delimiter

`\str_greplace_all:Nnn` nor the replacement can contain macro parameters or braces. The delimiter `\s__str_`

`\str_greplace_all:cnn` mark cannot appear in the string to edit so it is used in all cases. Some e-expansion is

`\str_replace_once:Nnn` unnecessary. There is no need to avoid losing braces nor to protect against expansion.

`\str_replace_once:cnn` The ending code is much simplified and does not need to hide in braces.

`\str_greplace_once:Nnn`

```

13754 \cs_new_protected:Npn \str_replace_once:Nnn
13755 { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_set:Nx }
13756 \cs_new_protected:Npn \str_greplace_once:Nnn
13757 { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_gset:Nx }
13758 \cs_new_protected:Npn \str_replace_all:Nnn
13759 { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_set:Nx }
13760 \cs_new_protected:Npn \str_greplace_all:Nnn
13761 { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_gset:Nx }
13762 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
13763 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
13764 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
13765 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
13766 \cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
13767 {
13768   \tl_if_empty:nTF {#4}
13769   {
13770     \msg_error:nne { kernel } { empty-search-pattern } {#5}
13771   }
13772   {

```

```

13773     \use:e
13774     {
13775         \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
13776         { \tl_to_str:N #3 }
13777         { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
13778     }
13779 }
13780 }
13781 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
13782 {
13783     \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
13784     #2 #3
13785     {
13786         \__str_replace_next:w
13787         #4
13788         \__str_use_none_delimit_by_s_stop:w
13789         #5
13790         \s__str_stop
13791     }
13792 }
13793 \cs_new_eq:NN \__str_replace_next:w ?

```

(End of definition for `\str_replace_all:Nnn` and others. These functions are documented on page 142.)

```

\str_remove_once:Nn Removal is just a special case of replacement.
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
13794 \cs_new_protected:Npn \str_remove_once:Nn #1#2
13795 { \str_replace_once:Nnn #1 {#2} { } }
13796 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
13797 { \str_greplace_once:Nnn #1 {#2} { } }
13798 \cs_generate_variant:Nn \str_remove_once:Nn { c }
13799 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End of definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 142.)

```

\str_remove_all:Nn Removal is just a special case of replacement.
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
13800 \cs_new_protected:Npn \str_remove_all:Nn #1#2
13801 { \str_replace_all:Nnn #1 {#2} { } }
13802 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
13803 { \str_greplace_all:Nnn #1 {#2} { } }
13804 \cs_generate_variant:Nn \str_remove_all:Nn { c }
13805 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End of definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 142.)

59.4 String comparisons

```

\str_if_empty_p:N More copy-paste!
\str_if_empty_p:c
\str_if_empty:NTF
\str_if_empty:cTF
\str_if_empty_p:n
\str_if_empty:nTF
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:NTF
\str_if_exist:cTF
13806 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
13807 { p , T , F , TF }
13808 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
13809 { p , T , F , TF }
13810 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N

```

```

13811 { p , T , F , TF }
13812 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
13813 { p , T , F , TF }
13814 \prg_new_eq_conditional:NNn \str_if_empty:n \tl_if_empty:n
13815 { p , T , F , TF }

```

(End of definition for `\str_if_empty:NTF`, `\str_if_empty:nTF`, and `\str_if_exist:NTF`. These functions are documented on page 136.)

`\__str_if_eq:nn` String comparisons rely on the primitive `\(pdf)strcmp`, so we define a new name for it.

```

13816 \cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D

```

(End of definition for `\__str_if_eq:nn`.)

`\str_compare_p:nNn` Simply rely on `\__str_if_eq:nn`, which expands to -1, 0 or 1. The `ee` version is created directly because it is more efficient.

`\str_compare_p:eNe`
`\str_compare:nNnTF`
`\str_compare:eNeTF`

```

13817 \prg_new_conditional:Npnn \str_compare:nNn #1#2#3 { p , T , F , TF }
13818 {
13819   \if_int_compare:w
13820     \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#3} }
13821     #2 \c_zero_int
13822     \prg_return_true: \else: \prg_return_false: \fi:
13823 }
13824 \prg_new_conditional:Npnn \str_compare:eNe #1#2#3 { p , T , F , TF }
13825 {
13826   \if_int_compare:w \__str_if_eq:nn {#1} {#3} #2 \c_zero_int
13827   \prg_return_true: \else: \prg_return_false: \fi:
13828 }

```

(End of definition for `\str_compare:nNnTF`. This function is documented on page 138.)

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore makes life a bit clearer. The `nn` and `ee` versions are created directly as this is most efficient. Since `\__str_if_eq:nn` will expand to 0 as an explicit character with category 12 if the two lists match (and either -1 or 1 if they don't) we can use `\if:w` here which is faster than using `\if_int_compare:w`.

```

13829 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
13830 {
13831   \if:w 0 \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
13832   \prg_return_true: \else: \prg_return_false: \fi:
13833 }
13834 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
13835 { V , v , o , nV , no , VV , nv } { p , T , F , TF }
13836 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
13837 {
13838   \if:w 0 \__str_if_eq:nn {#1} {#2}
13839   \prg_return_true: \else: \prg_return_false: \fi:
13840 }

```

(End of definition for `\str_if_eq:nnTF`. This function is documented on page 136.)

`\str_if_eq_p:NN` Note that `\str_if_eq:NNTF` is different from `\tl_if_eq:NNTF` because it needs to ignore category codes.

`\str_if_eq_p:Nc`
`\str_if_eq_p:cN`
`\str_if_eq_p:cc`
`\str_if_eq:NNTF`
`\str_if_eq:NcTF`
`\str_if_eq:cNTF`
`\str_if_eq:ccTF`

```

13841 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
13842 {

```

```

13843     \if:w 0 \__str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
13844     \prg_return_true: \else: \prg_return_false: \fi:
13845   }
13846   \prg_generate_conditional_variant:Nnn \str_if_eq:NN
13847   { c , Nc , cc } { T , F , TF , p }

```

(End of definition for `\str_if_eq:NNTF`. This function is documented on page 136.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.
`\str_if_in:cnTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

13848   \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
13849   {
13850     \use:e
13851     { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
13852     { \prg_return_true: } { \prg_return_false: }
13853   }
13854   \prg_generate_conditional_variant:Nnn \str_if_in:Nn
13855   { c } { T , F , TF }
13856   \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
13857   {
13858     \use:e
13859     { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
13860     { \prg_return_true: } { \prg_return_false: }
13861   }

```

(End of definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 136.)

`\str_case:nn` The aim here is to allow the case statement to be evaluated using a known number of
`\str_case:Vn` expansion steps (two), and without needing to use an explicit “end of recursion” marker.
`\str_case:on` That is achieved by using the test input as the final case, as this is always true. The
`\str_case:en` trick is then to tidy up the output such that the appropriate case code plus either the
`\str_case:nV` true or false branch code is inserted.

```

13862   \cs_new:Npn \str_case:nn #1#2
13863   {
13864     \exp:w
13865     \__str_case:nnTF {#1} {#2} { } { }
13866   }
13867   \cs_new:Npn \str_case:nnT #1#2#3
13868   {
13869     \exp:w
13870     \__str_case:nnTF {#1} {#2} {#3} { }
13871   }
13872   \cs_new:Npn \str_case:nnF #1#2
13873   {
13874     \exp:w
13875     \__str_case:nnTF {#1} {#2} { }
13876   }
13877   \cs_new:Npn \str_case:nnTF #1#2
13878   {
13879     \exp:w
13880     \__str_case:nnTF {#1} {#2}
13881   }

```

```

13882 \cs_new:Npn \__str_case:nnTF #1#2#3#4
13883 { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13884 \cs_generate_variant:Nn \str_case:nn { V , o , e , nV , nv , ne }
13885 \prg_generate_conditional_variant:Nnn \str_case:nn
13886 { V , o , e , nV , nv , ne } { T , F , TF }
13887 \cs_new_eq:NN \str_case:Nn \str_case:Vn
13888 \cs_new_eq:NN \str_case:NnT \str_case:VnT
13889 \cs_new_eq:NN \str_case:NnF \str_case:VnF
13890 \cs_new_eq:NN \str_case:NnTF \str_case:VnTF
13891 \cs_new:Npn \__str_case:nw #1#2#3
13892 {
13893   \str_if_eq:nnTF {#1} {#2}
13894   { \__str_case_end:nw {#3} }
13895   { \__str_case:nw {#1} }
13896 }
13897 \cs_new:Npn \str_case_e:nn #1#2
13898 {
13899   \exp:w
13900   \__str_case_e:nnTF {#1} {#2} { } { }
13901 }
13902 \cs_new:Npn \str_case_e:nnT #1#2#3
13903 {
13904   \exp:w
13905   \__str_case_e:nnTF {#1} {#2} {#3} { }
13906 }
13907 \cs_new:Npn \str_case_e:nnF #1#2
13908 {
13909   \exp:w
13910   \__str_case_e:nnTF {#1} {#2} { }
13911 }
13912 \cs_new:Npn \str_case_e:nnTF #1#2
13913 {
13914   \exp:w
13915   \__str_case_e:nnTF {#1} {#2}
13916 }
13917 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
13918 { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13919 \cs_generate_variant:Nn \str_case_e:nn { e }
13920 \prg_generate_conditional_variant:Nnn \str_case_e:nn { e } { T , F , TF }
13921 \cs_new:Npn \__str_case_e:nw #1#2#3
13922 {
13923   \str_if_eq:eeTF {#1} {#2}
13924   { \__str_case_end:nw {#3} }
13925   { \__str_case_e:nw {#1} }
13926 }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare \s__str_mark and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \s__str_mark and so #4 is the **false** code (the **true** code is mopped up by #3).

```

13927 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop

```

```
13928 { \exp_end: #1 #4 }
```

(End of definition for `\str_case:nnTF` and others. These functions are documented on page 137.)

59.5 Mapping over strings

```
\str_map_function:NN
\str_map_function:cN
\str_map_function:nN
\str_map_inline:Nn
\str_map_inline:cN
\str_map_inline:nn
\str_map_variable:NNn
\str_map_variable:cNn
\str_map_variable:nNn
\str_map_break:
\str_map_break:n
__str_map_function:w
__str_map_function:nn
__str_map_inline:NN
__str_map_variable:NnN
```

The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `\__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `\__str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `\__str_map_function:nn`, which passes the space to `#1` and calls `\__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

```
13929 \cs_new:Npn \str_map_function:nN #1#2
13930 {
13931   \exp_after:wN \__str_map_function:w
13932   \exp_after:wN \__str_map_function:nn \exp_after:wN #2
13933   \__kernel_tl_to_str:w {#1}
13934   \q__str_recursion_tail ? ~
13935   \prg_break_point:Nn \str_map_break: { }
13936 }
13937 \cs_new:Npn \str_map_function:NN
13938 { \exp_args:No \str_map_function:nN }
13939 \cs_new:Npn \__str_map_function:w #1 ~
13940 { #1 { ~ { ~ } } \__str_map_function:w } }
13941 \cs_new:Npn \__str_map_function:nn #1#2
13942 {
13943   \if_meaning:w \q__str_recursion_tail #2
13944   \exp_after:wN \str_map_break:
13945   \fi:
13946   #1 #2 \__str_map_function:nn {#1}
13947 }
13948 \cs_generate_variant:Nn \str_map_function:NN { c }
13949 \cs_new_protected:Npn \str_map_inline:nn #1#2
13950 {
13951   \int_gincr:N \g__kernel_prg_map_int
13952   \cs_gset_protected:cpn
13953   { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
13954   \use:e
13955   {
13956     \exp_not:N \__str_map_inline:NN
13957     \exp_not:c { __str_map_ \int_use:N \g__kernel_prg_map_int :w }
13958     \__kernel_str_to_other_fast:n {#1}
13959   }
13960   \q__str_recursion_tail
```

```

13961     \prg_break_point:Nn \str_map_break:
13962     { \int_gdecr:N \g__kernel_prg_map_int }
13963   }
13964   \cs_new_protected:Npn \str_map_inline:Nn
13965     { \exp_args:No \str_map_inline:nn }
13966   \cs_generate_variant:Nn \str_map_inline:Nn { c }
13967   \cs_new:Npn \__str_map_inline:NN #1#2
13968     {
13969       \__str_if_recursion_tail_break:NN #2 \str_map_break:
13970       \exp_args:No #1 { \token_to_str:N #2 }
13971       \__str_map_inline:NN #1
13972     }
13973   \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
13974     {
13975       \use:e
13976       {
13977         \exp_not:n { \__str_map_variable:NnN #2 {#3} }
13978         \__kernel_str_to_other_fast:n {#1}
13979       }
13980       \q__str_recursion_tail
13981       \prg_break_point:Nn \str_map_break: { }
13982     }
13983   \cs_new_protected:Npn \str_map_variable:NNn
13984     { \exp_args:No \str_map_variable:nNn }
13985   \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
13986     {
13987       \__str_if_recursion_tail_break:NN #3 \str_map_break:
13988       \str_set:Nn #1 {#3}
13989       \use:n {#2}
13990       \__str_map_variable:NnN #1 {#2}
13991     }
13992   \cs_generate_variant:Nn \str_map_variable:NNn { c }
13993   \cs_new:Npn \str_map_break:
13994     { \prg_map_break:Nn \str_map_break: { } }
13995   \cs_new:Npn \str_map_break:n
13996     { \prg_map_break:Nn \str_map_break: }

```

(End of definition for \str_map_function:NN and others. These functions are documented on page 138.)

\str_map_tokens:Nn Uses an auxiliary of \str_map_function:NN.

```

\str_map_tokens:cn 13997 \cs_new:Npn \str_map_tokens:nn #1#2
\str_map_tokens:nn 13998 {
13999   \exp_args:Nno \use:nn
14000   { \__str_map_function:w \__str_map_function:nn {#2} }
14001   { \__kernel_tl_to_str:w {#1} }
14002   \q__str_recursion_tail ? ~
14003   \prg_break_point:Nn \str_map_break: { }
14004 }
14005 \cs_new:Npn \str_map_tokens:Nn { \exp_args:No \str_map_tokens:nn }
14006 \cs_generate_variant:Nn \str_map_tokens:NNn { c }

```

(End of definition for \str_map_tokens:Nn and \str_map_tokens:nn. These functions are documented on page 138.)

59.6 Accessing specific characters in a string

`\__kernel_str_to_other:n` First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. `\__str_to_other_loop:w` The end is detected when `\__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `\__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

14007 \cs_new:Npn \__kernel_str_to_other:n #1
14008 {
14009     \exp_after:wN \__str_to_other_loop:w
14010     \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
14011 }
14012 \group_begin:
14013 \tex_lccode:D '\* = '\ %
14014 \tex_lccode:D '\A = '\A %
14015 \tex_lowercase:D
14016 {
14017     \group_end:
14018     \cs_new:Npn \__str_to_other_loop:w
14019         #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
14020     {
14021         \if_meaning:w A #8
14022         \__str_to_other_end:w
14023         \fi:
14024         \__str_to_other_loop:w
14025         #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
14026     }
14027     \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
14028     { \fi: #2 }
14029 }
```

(End of definition for `\__kernel_str_to_other:n`, `\__str_to_other_loop:w`, and `\__str_to_other_end:w`.)

`\__kernel_str_to_other_fast:n` The difference with `\__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable. `\__kernel_str_to_other_fast_loop:w` `\__str_to_other_fast_end:w`

```

14030 \cs_new:Npn \__kernel_str_to_other_fast:n #1
14031 {
14032     \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
14033     A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
14034 }
14035 \group_begin:
14036 \tex_lccode:D '\* = '\ %
14037 \tex_lccode:D '\A = '\A %
14038 \tex_lowercase:D
14039 {
14040     \group_end:
14041     \cs_new:Npn \__str_to_other_fast_loop:w
14042         #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
14043     {
14044         \if_meaning:w A #9
14045         \__str_to_other_fast_end:w
14046         \fi:
```



```

14047         #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
14048         \_str\_to\_other\_fast\_loop:w *
14049     }
14050     \cs\_new:Npn \_str\_to\_other\_fast\_end:w #1 * A #2 \s\_str\_stop {#1}
14051 }

```

(End of definition for `\_kernel\_str\_to\_other\_fast:n`, `\_kernel\_str\_to\_other\_fast\_loop:w`, and `\_str\_to\_other\_fast\_end:w`.)

`\_str\_sep:`

```

14052 \cs\_new\_eq:NN \_str\_sep: \_kernel\_int\_sep:

```

(End of definition for `\_str\_sep:`.)

`\str\_item:Nn` The `\str\_item:n` hands its argument with spaces escaped to `\_str\_item:n`, and `\str\_item:cn` makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str\_item\_ignore\_spaces:n` function does not escape spaces, which `\str\_item\_ignore\_spaces:n` are thus ignored by `\_str\_item:n` since everything else is done with undelimited arguments. Evaluate the `<index>` argument `#2` and count characters in the string, passing those two numbers to `\_str\_item:w` for further analysis. If the `<index>` is negative, shift it by the `<count>` to know the how many character to discard, and if that is still negative give an empty result. If the `<index>` is larger than the `<count>`, give an empty result, and otherwise discard `<index> - 1` characters before returning the following one. The shift by `-1` is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the `<index>` is zero.

```

14053 \cs\_new:Npn \str\_item:Nn { \exp\_args:No \str\_item:n }
14054 \cs\_generate\_variant:Nn \str\_item:Nn { c }
14055 \cs\_new:Npn \str\_item:n #1#2
14056 {
14057     \exp\_args:Nf \tl\_to\_str:n
14058     {
14059         \exp\_args:Nf \_str\_item:n
14060         { \_kernel\_str\_to\_other:n {#1} } {#2}
14061     }
14062 }
14063 \cs\_new:Npn \str\_item\_ignore\_spaces:n #1
14064 { \exp\_args:No \_str\_item:n { \tl\_to\_str:n {#1} } }
14065 \cs\_new:Npn \_str\_item:n #1#2
14066 {
14067     \exp\_after:wN \_str\_item:w
14068     \int\_value:w \int\_eval:n {#2} \exp\_after:wN \_str\_sep:
14069     \int\_value:w \_str\_count:n {#1} \_str\_sep:
14070     #1 \s\_str\_stop
14071 }
14072 \cs\_new:Npn \_str\_item:w #1 \_str\_sep: #2 \_str\_sep:
14073 {
14074     \int\_compare:nNnTF {#1} < 0
14075     {
14076         \int\_compare:nNnTF {#1} < {-#2}
14077         { \_str\_use\_none\_delimit\_by\_s\_stop:w }
14078         {
14079             \exp\_after:wN \_str\_use\_i\_delimit\_by\_s\_stop:nw
14080             \exp:w \exp\_after:wN \_str\_skip\_exp\_end:w
14081             \int\_value:w \int\_eval:n { #1 + #2 } \_str\_sep:

```

```

14082     }
14083   }
14084   {
14085     \int_compare:nNnTF {#1} > {#2}
14086     { \__str_use_none_delimit_by_s_stop:w }
14087     {
14088       \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
14089       \exp:w \__str_skip_exp_end:w #1 \__str_sep: { }
14090     }
14091   }
14092 }

```

(End of definition for \str_item:Nn and others. These functions are documented on page 141.)

<pre> __str_skip_exp_end:w __str_skip_loop:wNNNNNNNN __str_skip_end:w __str_skip_end:NNNNNNNN </pre>	Removes <code>max(#1,0)</code> characters from the input stream, and then leaves <code>\exp_end:</code>. This should be expanded using <code>\exp:w</code>. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the <code>\if_case:w</code> construction leaves between 0 and 8 times the <code>\or:</code> control sequence, and those <code>\or:</code> become arguments of <code>__str_skip_end:NNNNNNNN</code>. If the number of characters to remove is 6, say, then there are two <code>\or:</code> left, and the 8 arguments of <code>__str_skip_end:NNNNNNNN</code> are the two <code>\or:</code>, and 6 characters from the input stream, exactly what we wanted to remove. Then close the <code>\if_case:w</code> conditional with <code>\fi:</code>, and stop the initial expansion with <code>\exp_end:</code> (see places where <code>__str_skip_exp_end:w</code> is called).

```

14093 \cs_new:Npn \__str_skip_exp_end:w #1 \__str_sep:
14094 {
14095   \if_int_compare:w #1 > 8 \exp_stop_f:
14096     \exp_after:wN \__str_skip_loop:wNNNNNNNN
14097   \else:
14098     \exp_after:wN \__str_skip_end:w
14099     \int_value:w \int_eval:w
14100   \fi:
14101   #1 \__str_sep:
14102 }
14103 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1 \__str_sep: #2#3#4#5#6#7#8#9
14104 {
14105   \exp_after:wN \__str_skip_exp_end:w
14106   \int_value:w \int_eval:n { #1 - 8 } \__str_sep:
14107 }
14108 \cs_new:Npn \__str_skip_end:w #1 \__str_sep:
14109 {
14110   \exp_after:wN \__str_skip_end:NNNNNNNN
14111   \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
14112 }
14113 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

```

(End of definition for __str_skip_exp_end:w and others.)

<pre> \str_range:Nnn \str_range:nnn \str_range_ignore_spaces:nnn __str_range:nnn __str_range:w __str_range:nw </pre>	Sanitize the string. Then evaluate the arguments. At this stage we also decrement the <code><start index></code>, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```

14114 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
14115 \cs_generate_variant:Nn \str_range:Nnn { c }
14116 \cs_new:Npn \str_range:nnn #1#2#3
14117 {
14118   \exp_args:Nf \tl_to_str:n
14119   {
14120     \exp_args:Nf \__str_range:nnn
14121     { \__kernel_str_to_other:n {#1} } {#2} {#3}
14122   }
14123 }
14124 \cs_new:Npn \str_range_ignore_spaces:nnn #1
14125 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
14126 \cs_new:Npn \__str_range:nnn #1#2#3
14127 {
14128   \exp_after:wN \__str_range:w
14129   \int_value:w \__str_count:n {#1} \exp_after:wN \__str_sep:
14130   \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN \__str_sep:
14131   \int_value:w \int_eval:n {#3} \__str_sep:
14132   #1 \s__str_stop
14133 }
14134 \cs_new:Npn \__str_range:w #1 \__str_sep: #2 \__str_sep: #3 \__str_sep:
14135 {
14136   \exp_args:Nf \__str_range:nnw
14137   { \__str_range_normalize:nn {#2} {#1} }
14138   { \__str_range_normalize:nn {#3} {#1} }
14139 }
14140 \cs_new:Npn \__str_range:nnw #1#2
14141 {
14142   \exp_after:wN \__str_collect_delimit_by_q_stop:w
14143   \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN \__str_sep:
14144   \exp:w \__str_skip_exp_end:w #1 \__str_sep:
14145 }

```

(End of definition for `\str_range:Nnn` and others. These functions are documented on page 141.)

`\__str_range_normalize:nn` This function converts an `<index>` argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` #1 and the string count #2. If #1 is negative, replace it by $\#1 + \#2 + 1$, then limit to the range $[0, \#2]$.

```

14146 \cs_new:Npn \__str_range_normalize:nn #1#2
14147 {
14148   \int_eval:n
14149   {
14150     \if_int_compare:w #1 < \c_zero_int
14151       \if_int_compare:w #1 < -#2 \exp_stop_f:
14152       0
14153     \else:
14154       #1 + #2 + 1
14155     \fi:
14156   \else:
14157     \if_int_compare:w #1 < #2 \exp_stop_f:
14158     #1
14159   \else:
14160     #2

```

```

14161         \fi:
14162     \fi:
14163 }
14164 }

```

(End of definition for `\_str_range_normalize:nn`.)

`\_str_collect_delimit_by_q_stop:w` Collects `max(#1,0)` characters, and removes everything else until `\s\_str_stop`. This is somewhat similar to `\_str_skip_exp_end:w`, but accepts integer expression arguments. `\_str_collect_loop:wn` This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `\_str_collect_end:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by #1 characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

14165 \cs_new:Npn \_str_collect_delimit_by_q_stop:w #1 \_str_sep:
14166 { \_str_collect_loop:wn #1 \_str_sep: { } }
14167 \cs_new:Npn \_str_collect_loop:wn #1 \_str_sep:
14168 {
14169     \if_int_compare:w #1 > 7 \exp_stop_f:
14170     \exp_after:wN \_str_collect_loop:wnNNNNNNN
14171     \else:
14172     \exp_after:wN \_str_collect_end:wn
14173     \fi:
14174     #1 \_str_sep:
14175 }
14176 \cs_new:Npn \_str_collect_loop:wnNNNNNNN #1 \_str_sep: #2 #3#4#5#6#7#8#9
14177 {
14178     \exp_after:wN \_str_collect_loop:wn
14179     \int_value:w \int_eval:n { #1 - 7 } \_str_sep:
14180     { #2 #3#4#5#6#7#8#9 }
14181 }
14182 \cs_new:Npn \_str_collect_end:wn #1 \_str_sep:
14183 {
14184     \exp_after:wN \_str_collect_end:nnnnnnnnw
14185     \if_case:w \if_int_compare:w #1 > \c_zero_int
14186     #1 \else: 0 \fi: \exp_stop_f:
14187     \or: \or: \or: \or: \or: \or: \or: \fi:
14188 }
14189 \cs_new:Npn \_str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s\_str_stop
14190 { #1#2#3#4#5#6#7#8 }

```

(End of definition for `\_str_collect_delimit_by_q_stop:w` and others.)

59.7 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X(number)`, and that `(number)` is added to the sum of 9 that precedes, to adjust the result.

```

14191 \cs_new:Npn \str_count_spaces:N
14192 { \exp_args:No \str_count_spaces:n }
14193 \cs_generate_variant:Nn \str_count_spaces:N { c }
14194 \cs_new:Npn \str_count_spaces:n #1
14195 {

```

```

14196 \int_eval:n
14197 {
14198   \exp_after:wN \__str_count_spaces_loop:w
14199   \tl_to_str:n {#1} ~
14200   X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
14201   \s__str_stop
14202 }
14203 }
14204 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
14205 {
14206   \if_meaning:w X #9
14207   \__str_use_i_delimit_by_s_stop:nw
14208   \fi:
14209   9 + \__str_count_spaces_loop:w
14210 }

```

(End of definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `\__str_count_spaces_loop:w`. These functions are documented on page 140.)

`\str_count:N` To count characters in a string we could first escape all spaces using `\__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, `\str_count:n` sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, `loop`, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `\__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

`\str_count_ignore_spaces:n`
`\str_count_ignore_spaces:e`
`\__str_count:n`
`\__str_count_aux:n`
`\__str_count_loop:NNNNNNNN`

```

14211 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
14212 \cs_generate_variant:Nn \str_count:N { c }
14213 \cs_new:Npn \str_count:n #1
14214 {
14215   \__str_count_aux:n
14216   {
14217     \str_count_spaces:n {#1}
14218     + \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1}
14219   }
14220 }
14221 \cs_generate_variant:Nn \str_count:n { e }
14222 \cs_new:Npn \__str_count:n #1
14223 {
14224   \__str_count_aux:n
14225   { \__str_count_loop:NNNNNNNN #1 }
14226 }
14227 \cs_new:Npn \str_count_ignore_spaces:n #1
14228 {
14229   \__str_count_aux:n
14230   { \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1} }
14231 }
14232 \cs_generate_variant:Nn \str_count_ignore_spaces:n { e }
14233 \cs_new:Npn \__str_count_aux:n #1
14234 {
14235   \int_eval:n

```

```

14236     {
14237         #1
14238         { X 8 } { X 7 } { X 6 }
14239         { X 5 } { X 4 } { X 3 }
14240         { X 2 } { X 1 } { X 0 }
14241         \s__str_stop
14242     }
14243 }
14244 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
14245 {
14246     \if_meaning:w X #9
14247     \exp_after:wN \__str_use_none_delimit_by_s_stop:w
14248     \fi:
14249     9 + \__str_count_loop:NNNNNNNN
14250 }

```

(End of definition for `\str_count:N` and others. These functions are documented on page 140.)

59.8 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.
`\str_head:c` To circumvent the fact that TeX skips spaces when grabbing undelimited macro parameters, `\__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\__str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `\__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `\__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\__str_use_i_delimit_by_s_stop:nw`.

```

14251 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
14252 \cs_generate_variant:Nn \str_head:N { c }
14253 \cs_new:Npn \str_head:n #1
14254 {
14255     \exp_after:wN \__str_head:w
14256     \tl_to_str:n {#1}
14257     { { } } ~ \s__str_stop
14258 }
14259 \cs_new:Npn \__str_head:w #1 ~ %
14260 { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
14261 \cs_new:Npn \str_head_ignore_spaces:n #1
14262 {
14263     \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
14264     \tl_to_str:n {#1} { } \s__str_stop
14265 }

```

(End of definition for `\str_head:N` and others. These functions are documented on page 140.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N` `\if_charcode:w` `\scan_stop:.` This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:.` The auxiliary function then inserts the required `\fi:` to close the conditional, and
`\str_tail:c`
`\str_tail:n`
`\str_tail_ignore_spaces:n`
`\__str_tail_auxi:w`
`\__str_tail_auxii:w`

leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

14266 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
14267 \cs_generate_variant:Nn \str_tail:N { c }
14268 \cs_new:Npn \str_tail:n #1
14269 {
14270   \exp_after:wN \__str_tail_auxi:w
14271   \reverse_if:N \if_charcode:w
14272     \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
14273 }
14274 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
14275 \cs_new:Npn \str_tail_ignore_spaces:n #1
14276 {
14277   \exp_after:wN \__str_tail_auxii:w
14278   \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
14279 }
14280 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

```

(End of definition for `\str_tail:N` and others. These functions are documented on page 140.)

59.9 String manipulation

<pre> \str_casefold:n \str_casefold:V \str_lowercase:n \str_lowercase:f \str_uppercase:n \str_uppercase:f __str_change_case:nn __str_change_case_aux:nn __str_change_case_result:n __str_change_case_output:nw __str_change_case_output:fw __str_change_case_end:nw __str_change_case_loop:nw __str_change_case_space:n __str_change_case_char:nN __str_change_case_char_auxi:nN __str_change_case_char_auxii:nN __str_change_case_codepoint:nN __str_change_case_codepoint:nNN __str_change_case_codepoint:nNNN __str_change_case_char:nnn __str_change_case_char_aux:nnn __str_change_case_char:nnnnn </pre>	Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing. Similarly, for 8-bit engines the multi-byte information is shared. <pre> 14281 \cs_new:Npn \str_casefold:n #1 { __str_change_case:nn {#1} { casefold } } 14282 \cs_new:Npn \str_lowercase:n #1 { __str_change_case:nn {#1} { lowercase } } 14283 \cs_new:Npn \str_uppercase:n #1 { __str_change_case:nn {#1} { uppercase } } 14284 \cs_generate_variant:Nn \str_casefold:n { V } 14285 \cs_generate_variant:Nn \str_lowercase:n { f } 14286 \cs_generate_variant:Nn \str_uppercase:n { f } 14287 \cs_new:Npn __str_change_case:nn #1 14288 { 14289 \exp_after:wN __str_change_case_aux:nn \exp_after:wN 14290 { \tl_to_str:n {#1} } 14291 } 14292 \cs_new:Npn __str_change_case_aux:nn #1#2 14293 { 14294 __str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop 14295 __str_change_case_result:n { } 14296 } 14297 \cs_new:Npn __str_change_case_output:nw #1#2 __str_change_case_result:n #3 14298 { #2 __str_change_case_result:n { #3 #1 } } 14299 \cs_generate_variant:Nn __str_change_case_output:nw { f } 14300 \cs_new:Npn __str_change_case_end:wn #1 __str_change_case_result:n #2 14301 { \tl_to_str:n {#2} } 14302 \cs_new:Npn __str_change_case_loop:nw #1#2 \q__str_recursion_stop </pre>

```

14303 {
14304   \tl_if_head_is_space:nTF {#2}
14305     { \__str_change_case_space:n }
14306     { \__str_change_case_char:nN }
14307   {#1} #2 \q_str_recursion_stop
14308 }
14309 \exp_last_unbraced:NNNNo
14310 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
14311 {
14312   \__str_change_case_output:nw { ~ }
14313   \__str_change_case_loop:nw {#1}
14314 }
14315 \cs_new:Npn \__str_change_case_char:nN #1#2
14316 {
14317   \__str_if_recursion_tail_stop_do:Nn #2
14318     { \__str_change_case_end:wn }
14319   \__str_change_case_codepoint:nN {#1} #2
14320 }
14321 \if_int_compare:w 0
14322   \cs_if_exist:NT \tex_XeTeXversion:D { 1 }
14323   \cs_if_exist:NT \tex_luatexversion:D { 1 }
14324   > 0 \exp_stop_f:
14325   \cs_new:Npn \__str_change_case_codepoint:nN #1#2
14326     { \__str_change_case_char:fnn { \int_eval:n {'#2} } {#1} {#2} }
14327 \else:
14328   \cs_new:Npe \__str_change_case_codepoint:nN #1#2
14329   {
14330     \exp_not:N \int_compare:nNnTF {'#2} > { "80 }
14331     {
14332       \cs_if_exist:NTF \tex_pdftexversion:D
14333         { \exp_not:N \__str_change_case_char_auxi:nN }
14334         {
14335           \exp_not:N \int_compare:nNnTF {'#2} > { "FF }
14336           { \exp_not:N \__str_change_case_char_auxii:nN }
14337           { \exp_not:N \__str_change_case_char_auxi:nN }
14338         }
14339       }
14340     { \exp_not:N \__str_change_case_char_auxii:nN }
14341     {#1} #2
14342   }
14343   \cs_new:Npn \__str_change_case_char_auxi:nN #1#2
14344   {
14345     \int_compare:nNnTF {'#2} < { "E0 }
14346     { \__str_change_case_codepoint:nN }
14347     {
14348       \int_compare:nNnTF {'#2} < { "F0 }
14349       { \__str_change_case_codepoint:nNNN }
14350       { \__str_change_case_codepoint:nNNNNN }
14351     }
14352     {#1} #2
14353   }
14354   \cs_new:Npn \__str_change_case_char_auxii:nN #1#2
14355     { \__str_change_case_char:fnn { \int_eval:n {'#2} } {#1} {#2} }
14356   \cs_new:Npn \__str_change_case_codepoint:nN #1#2#3

```



```

14357     {
14358         \__str_change_case_char:fnn
14359         { \int_eval:n { ('#2 - "C0) * "40 + '#3 - "80 } }
14360         {#1} {#2#3}
14361     }
14362 \cs_new:Npn \__str_change_case_codepoint:nNNN #1#2#3#4
14363 {
14364     \__str_change_case_char:fnn
14365     {
14366         \int_eval:n
14367         { ('#2 - "E0) * "1000 + ('#3 - "80) * "40 + '#4 - "80 }
14368     }
14369     {#1} {#2#3#4}
14370 }
14371 \cs_new:Npn \__str_change_case_codepoint:nNNNN #1#2#3#4#5
14372 {
14373     \__str_change_case_char:fnn
14374     {
14375         \int_eval:n
14376         {
14377             ('#2 - "F0) * "40000
14378             + ('#3 - "80) * "1000
14379             + ('#4 - "80) * "40
14380             + '#5 - "80
14381         }
14382     }
14383     {#1} {#2#3#4#5}
14384 }
14385 \fi:
14386 \cs_new:Npn \__str_change_case_char:nnn #1#2#3
14387 {
14388     \__str_change_case_output:fw
14389     {
14390         \exp_args:Ne \__str_change_case_char_aux:nnn
14391         { \__kernel_codepoint_case:nn {#2} {#1} } {#1} {#3}
14392     }
14393     \__str_change_case_loop:nw {#2}
14394 }
14395 \cs_generate_variant:Nn \__str_change_case_char:nnn { f }
14396 \cs_new:Npn \__str_change_case_char_aux:nnn #1#2#3
14397 {
14398     \use:e { \__str_change_case_char:nnnnn #1 {#2} {#3} }
14399 }
14400 \cs_new:Npn \__str_change_case_char:nnnnn #1#2#3#4#5
14401 {
14402     \int_compare:nNnTF {#1} = {#4}
14403     { \tl_to_str:n {#5} }
14404     {
14405         \codepoint_str_generate:n {#1}
14406         \tl_if_blank:nF {#2}
14407         {
14408             \codepoint_str_generate:n {#2}
14409             \tl_if_blank:nF {#3}
14410             { \codepoint_str_generate:n {#3} }

```

```

14411     }
14412   }
14413 }

```

(End of definition for `\str_casefold:n` and others. These functions are documented on page 144.)

```

\str_mdfive_hash:n
\str_mdfive_hash:V 14414 \cs_new:Npn \str_mdfive_hash:n #1 { \tex_mdffivesum:D { \tl_to_str:n {#1} } }
\str_mdfive_hash:v 14415 \cs_generate_variant:Nn \str_mdfive_hash:n { V , v , o }
\str_mdfive_hash:o 14416 \cs_new:Npn \str_mdfive_hash:e #1 { \tex_mdffivesum:D {#1} }
\str_mdfive_hash:e

```

(End of definition for `\str_mdfive_hash:n`. This function is documented on page 144.)

```

\c_ampersand_str For all of those strings, use \cs_to_str:N to get characters with the correct category
\c_atsign_str    code without worries
\c_backslash_str 14417 \str_const:Ne \c_ampersand_str { \cs_to_str:N \& }
\c_left_brace_str 14418 \str_const:Ne \c_atsign_str { \cs_to_str:N \@ }
\c_right_brace_str 14419 \str_const:Ne \c_backslash_str { \cs_to_str:N \\ }
\c_circumflex_str 14420 \str_const:Ne \c_left_brace_str { \cs_to_str:N \{ }
\c_colon_str      14421 \str_const:Ne \c_right_brace_str { \cs_to_str:N \} }
\c_dollar_str     14422 \str_const:Ne \c_circumflex_str { \cs_to_str:N \^ }
\c_hash_str       14423 \str_const:Ne \c_colon_str { \cs_to_str:N \: }
\c_percent_str    14424 \str_const:Ne \c_dollar_str { \cs_to_str:N \$ }
\c_tilde_str      14425 \str_const:Ne \c_hash_str { \cs_to_str:N \# }
\c_underscore_str 14426 \str_const:Ne \c_percent_str { \cs_to_str:N \% }
\c_zero_str       14427 \str_const:Ne \c_tilde_str { \cs_to_str:N \~ }
                  14428 \str_const:Ne \c_underscore_str { \cs_to_str:N \_ }
                  14429 \str_const:Ne \c_zero_str { 0 }

```

(End of definition for `\c_ampersand_str` and others. These variables are documented on page 145.)

```

\c_empty_str An empty string is simply an empty token list.
14430 \cs_new_eq:NN \c_empty_str \c_empty_tl

```

(End of definition for `\c_empty_str`. This variable is documented on page 145.)

```

\l_tmpa_str Scratch strings.
\l_tmpb_str 14431 \str_new:N \l_tmpa_str
\g_tmpa_str 14432 \str_new:N \l_tmpb_str
\g_tmpb_str 14433 \str_new:N \g_tmpa_str
14434 \str_new:N \g_tmpb_str

```

(End of definition for `\l_tmpa_str` and others. These variables are documented on page 145.)

59.10 Viewing strings

```

\str_show:n Displays a string on the terminal.
\str_show:N 14435 \cs_new_eq:NN \str_show:n \tl_show:n
\str_show:c 14436 \cs_new_protected:Npn \str_show:N #1
\str_log:n   14437 {
\str_log:N   14438   \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
\str_log:c   14439   { \tl_show:N #1 }
14440 }
14441 \cs_generate_variant:Nn \str_show:N { c }

```

```

14442 \cs_new_eq:NN \str_log:n \tl_log:n
14443 \cs_new_protected:Npn \str_log:N #1
14444 {
14445     \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
14446     { \tl_log:N #1 }
14447 }
14448 \cs_generate_variant:Nn \str_log:N { c }

```

(End of definition for \str_show:n and others. These functions are documented on page [144](#).)

```

14449 \</code>

```

Chapter 60

l3str-convert implementation

14450 $\langle *code \rangle$

14451 $\langle @@=str \rangle$

60.1 Helpers

60.1.1 Variables and constants

$\_str\_tmp:w$ Internal scratch space for some functions.
 $\_l\_str\_tmp\_tl$ 14452 $\backslash cs\_new\_protected:Npn \_str\_tmp:w \{ \}$
14453 $\backslash tl\_new:N \_l\_str\_tmp\_tl$

(End of definition for $\_str\_tmp:w$ and $\_l\_str\_tmp\_tl$.)

$\_g\_str\_result\_tl$ The $\_g\_str\_result\_tl$ variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

14454 $\backslash tl\_new:N \_g\_str\_result\_tl$

(End of definition for $\_g\_str\_result\_tl$.)

$\_c\_str\_replacement\_char\_int$ When converting, invalid bytes are replaced by the Unicode replacement character "FFFD.

14455 $\backslash int\_const:Nn \_c\_str\_replacement\_char\_int \{ "FFFD \}$

(End of definition for $\_c\_str\_replacement\_char\_int$.)

$\_c\_str\_max\_byte\_int$ The maximal byte number.
14456 $\backslash int\_const:Nn \_c\_str\_max\_byte\_int \{ 255 \}$

(End of definition for $\_c\_str\_max\_byte\_int$.)

$\_s\_str$ Internal scan marks.

14457 $\backslash scan\_new:N \_s\_str$

(End of definition for $\_s\_str$.)

`\q__str_nil` Internal quarks.

```
14458 \quark_new:N \q__str_nil
```

(End of definition for \q__str_nil.)

`\g__str_alias_prop` To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```
14459 \prop_new:N \g__str_alias_prop
14460 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
14461 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
14462 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
14463 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
14464 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
14465 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
14466 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
14467 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
14468 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
14469 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
14470 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
14471 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
14472 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
14473 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
14474 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
14475 \bool_lazy_any:nTF
14476 {
14477   \sys_if_engine luatex_p:
14478   \sys_if_engine xetex_p:
14479 }
14480 {
14481   \prop_gput:Nnn \g__str_alias_prop { default } { }
14482 }
14483 {
14484   \prop_gput:Nnn \g__str_alias_prop { default } { utf8 }
14485 }
```

(End of definition for \g__str_alias_prop.)

`\g__str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
14486 \bool_new:N \g__str_error_bool
```

(End of definition for \g__str_error_bool.)

`\l__str_byte_flag` `\l__str_error_flag` Conversions from one *<encoding>*/*<escaping>* pair to another are done within e-expanding assignments. Errors are signaled by raising the relevant flag.

```
14487 \flag_new:N \l__str_byte_flag
14488 \flag_new:N \l__str_error_flag
```

(End of definition for \l__str_byte_flag and \l__str_error_flag.)

60.2 String conditionals

`\_str_if_contains_char:NnT` `\_str_if_contains_char:nnTF {⟨token list⟩} ⟨char⟩`
`\_str_if_contains_char:NnTF` Expects the `⟨token list⟩` to be an `⟨other string⟩`: the caller is responsible for
`\_str_if_contains_char:nnTF` ensuring that no (too-)special catcodes remain. Loop over the characters of the string,
 `\_str_if_contains_char_aux:nn` comparing character codes. The loop is broken if character codes match. Otherwise we
 `\_str_if_contains_char_aux:nN` return “false”.
 `\_str_if_contains_char_true:`

```

14489 \prg_new_conditional:Npnn \_str_if_contains_char:Nn #1#2 { T , TF }
14490 {
14491   \exp_after:wN \_str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
14492   { \prg_break:n { ? \fi: } }
14493   \prg_break_point:
14494   \prg_return_false:
14495 }
14496 \cs_new:Npn \_str_if_contains_char_aux:nn #1#2
14497 { \_str_if_contains_char_auxi:nN {#2} #1 }
14498 \prg_new_conditional:Npnn \_str_if_contains_char:nn #1#2 { TF }
14499 {
14500   \_str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
14501   \prg_break_point:
14502   \prg_return_false:
14503 }
14504 \cs_new:Npn \_str_if_contains_char_auxi:nN #1#2
14505 {
14506   \if_charcode:w #1 #2
14507   \exp_after:wN \_str_if_contains_char_true:
14508   \fi:
14509   \_str_if_contains_char_auxi:nN {#1}
14510 }
14511 \cs_new:Npn \_str_if_contains_char_true:
14512 { \prg_break:n { \prg_return_true: \use_none:n } }

```

(End of definition for `\_str_if_contains_char:NnT` and others.)

`\_str_octal_use:NTF` `\_str_octal_use:NTF {⟨token⟩} {⟨true code⟩} {⟨false code⟩}`
 If the `⟨token⟩` is an octal digit, it is left in the input stream, *followed* by the `⟨true code⟩`. Otherwise, the `⟨false code⟩` is left in the input stream.

T_EXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. T_EX dutifully detects

octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is ‘1#1, greater than 1. Otherwise, the right-hand side stops as ‘1, and the conditional takes the false branch.

```

14513 \prg_new_conditional:Npnn \_str_octal_use:N #1 { TF }
14514 {
14515   \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
14516   #1 \prg_return_true:
14517   \else:
14518   \prg_return_false:
14519   \fi:
14520 }

```

(End of definition for `\_str_octal_use:NTF`.)

`\__str_hexadecimal_use:N`TF TeX detects uppercase hexadecimal digits for us (see `\__str_octal_use:N`TF), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

14521 \prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
14522 {
14523   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
14524     #1 \prg_return_true:
14525   \else:
14526     \if_case:w \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
14527       A
14528     \or: B
14529     \or: C
14530     \or: D
14531     \or: E
14532     \or: F
14533   \else:
14534     \prg_return_false:
14535     \exp_after:wN \use_none:n
14536   \fi:
14537   \prg_return_true:
14538 \fi:
14539 }

```

(End of definition for `\__str_hexadecimal_use:N`TF.)

60.3 Conversions

60.3.1 Producing one byte or character

`\c__str_byte_0_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

14540 \group_begin:
14541   \__kernel_tl_set:Nx \l__str_tmp_tl { \tl_to_str:n { 0123456789ABCDEF } }
14542   \tl_map_inline:Nn \l__str_tmp_tl
14543   {
14544     \tl_map_inline:Nn \l__str_tmp_tl
14545     {
14546       \tl_const:ce { c__str_byte_ \int_eval:n {"#1##1} _tl }
14547       { \char_generate:nn { "#1##1 } { 12 } #1 ##1 }
14548     }
14549   }
14550 \group_end:
14551 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }

```

(End of definition for `\c__str_byte_0_tl` and others.)

`\__str_output_byte:n` Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_⟨number⟩_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```

14552 \cs_new:Npn \__str_output_byte:n #1
14553 { \__str_output_byte:w #1 \__str_output_end: }
14554 \cs_new:Npn \__str_output_byte:w
14555 {
14556   \exp_after:wN \exp_after:wN
14557   \exp_after:wN \use_i:nnn
14558   \cs:w c__str_byte_ \int_eval:w
14559 }
14560 \cs_new:Npn \__str_output_hexadecimal:n #1
14561 {
14562   \exp_after:wN \exp_after:wN
14563   \exp_after:wN \use_none:n
14564   \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
14565 }
14566 \cs_new:Npn \__str_output_end:
14567 { \scan_stop: _tl \cs_end: }

```

(End of definition for `\__str_output_byte:n` and others.)

`\__str_output_byte_pair_be:n` Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.
`\__str_output_byte_pair_le:n`
`\__str_output_byte_pair:nnN`

```

14568 \cs_new:Npn \__str_output_byte_pair_be:n #1
14569 {
14570   \exp_args:Nf \__str_output_byte_pair:nnN
14571   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
14572 }
14573 \cs_new:Npn \__str_output_byte_pair_le:n #1
14574 {
14575   \exp_args:Nf \__str_output_byte_pair:nnN
14576   { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
14577 }
14578 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
14579 {
14580   #3
14581   { \__str_output_byte:n { #1 } }
14582   { \__str_output_byte:n { #2 - #1 * "100 } }
14583 }

```

(End of definition for `\__str_output_byte_pair_be:n`, `\__str_output_byte_pair_le:n`, and `\__str_output_byte_pair:nnN`.)

60.3.2 Mapping functions for conversions

`\__str_convert_gmap:N` This maps the function #1 over all characters in `\g__str_result_tl`, which should be a
`\__str_convert_gmap_loop:NN` byte string in most cases, sometimes a native string.

```

14584 \cs_new_protected:Npn \__str_convert_gmap:N #1
14585 {
14586   \__kernel_tl_gset:Nx \g__str_result_tl
14587   {
14588     \exp_after:wN \__str_convert_gmap_loop:NN
14589     \exp_after:wN #1
14590     \g__str_result_tl { ? \prg_break: }
14591     \prg_break_point:
14592   }

```



```

14593     }
14594 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
14595 {
14596     \use_none:n #2
14597     #1#2
14598     \__str_convert_gmap_loop:NN #1
14599 }

```

(End of definition for __str_convert_gmap:N and __str_convert_gmap_loop:NN.)

```

\__str_convert_gmap_internal:N
\__str_convert_gmap_internal_loop:Nw

```

This maps the function #1 over all character codes in \g__str_result_tl, which must be in the internal representation.

```

14600 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
14601 {
14602     \__kernel_tl_gset:Nx \g__str_result_tl
14603     {
14604         \exp_after:wN \__str_convert_gmap_internal_loop:Nww
14605         \exp_after:wN #1
14606         \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
14607         \prg_break_point:
14608     }
14609 }
14610 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
14611 {
14612     \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
14613     #1 {#3}
14614     \__str_convert_gmap_internal_loop:Nww #1
14615 }

```

(End of definition for __str_convert_gmap_internal:N and __str_convert_gmap_internal_loop:Nw.)

60.3.3 Error-reporting during conversion

```

\__str_if_flag_error:Nne
\__str_if_flag_no_error:Nne

```

When converting using the function \str_set_convert:Nnnn, errors should be reported to the user after each step in the conversion. Errors are signaled by raising some flag (typically @@_error), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions \str_set_convert:NnnnTF, errors should be suppressed. This is done by changing __str_if_flag_error:Nne into __str_if_flag_no_error:Nne locally.

```

14616 \cs_new_protected:Npn \__str_if_flag_error:Nne #1
14617 {
14618     \flag_if_raised:NTF #1
14619     { \msg_error:nne { str } }
14620     { \use_none:nn }
14621 }
14622 \cs_new_protected:Npn \__str_if_flag_no_error:Nne #1#2#3
14623 { \flag_if_raised:NT #1 { \bool_gset_true:N \g__str_error_bool } }

```

(End of definition for __str_if_flag_error:Nne and __str_if_flag_no_error:Nne.)

```

\__str_if_flag_times:NT

```

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

14624 \cs_new:Npn \__str_if_flag_times:NT #1#2
14625 { \flag_if_raised:NT #1 { #2~(x \flag_height:N #1 ) } }

```

(End of definition for __str_if_flag_times:NT.)

60.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of T_EX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

$\langle \text{bytes} \rangle \backslash s\_str \langle \text{Unicode code point} \rangle \backslash s\_str$

where we have collected the $\langle \text{bytes} \rangle$ which combined to form this particular Unicode character, and the $\langle \text{Unicode code point} \rangle$ is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

```

\str_set_convert:Nnnn
\str_gset_convert:Nnnn
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
\__str_convert:nNNnnn

```

The input string is stored in $\backslash g\_str\_result\_tl$, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on $\backslash g\_str\_result\_tl$. Errors are silenced for the conditional functions by redefining $\backslash \_str\_if\_flag\_error:Nne$ locally.

```

14626 \cs_new_protected:Npn \str_set_convert:Nnnn
14627 { \__str_convert:nNNnnn { } \tl_set_eq:NN }
14628 \cs_new_protected:Npn \str_gset_convert:Nnnn
14629 { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
14630 \prg_new_protected_conditional:Npnn
14631 \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
14632 {
14633   \bool_gset_false:N \g__str_error_bool
14634   \__str_convert:nNNnnn
14635   { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
14636   \tl_set_eq:NN #1 {#2} {#3} {#4}
14637   \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14638 }
14639 \prg_new_protected_conditional:Npnn
14640 \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }

```

```

14641 {
14642     \bool_gset_false:N \g__str_error_bool
14643     \__str_convert:nNNnnn
14644     { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
14645     \tl_gset_eq:NN #1 {#2} {#3} {#4}
14646     \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14647 }
14648 \cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
14649 {
14650     \group_begin:
14651     #1
14652     \__kernel_tl_gset:Nx \g__str_result_tl { \__kernel_str_to_other_fast:n {#4} }
14653     \exp_after:wN \__str_convert:wwwnn
14654     \tl_to_str:n {#5} /// \s__str_stop
14655     { decode } { unescape }
14656     \prg_do_nothing:
14657     \__str_convert_decode_:
14658     \exp_after:wN \__str_convert:wwwnn
14659     \tl_to_str:n {#6} /// \s__str_stop
14660     { encode } { escape }
14661     \use_ii_i:nn
14662     \__str_convert_encode_:
14663     \__kernel_tl_gset:Nx \g__str_result_tl
14664     { \tl_to_str:V \g__str_result_tl }
14665     \group_end:
14666     #2 #3 \g__str_result_tl
14667 }

```

(End of definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 148.)

`\__str_convert:wwwnn` The task of `\__str_convert:wwwnn` is to split `<encoding>/<escaping>` pairs into their components, `#1` and `#2`. Calls to `\__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- `#1` is the encoding conversion function;
- `#2` is the escaping function;
- `#3` is the escaping name for use in an error message;
- `#4` is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- `#5` is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do `#2#1` in this order, and in the encoding/escaping phase, the order should be reversed: `#4#2#1` does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, `#1`, should be performed.

```

14668 \cs_new_protected:Npn \__str_convert:wwwnn
14669     #1 / #2 // #3 \s__str_stop #4#5
14670 {

```

```

14671     \__str_convert:nnn {enc} {#4} {#1}
14672     \__str_convert:nnn {esc} {#5} {#2}
14673     \exp_args:Ncc \__str_convert:NNnNN
14674     { \__str_convert_#4_#1: } { \__str_convert_#5_#2: } {#2}
14675 }
14676 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
14677 {
14678     \if_meaning:w #1 #5
14679     \tl_if_empty:nF {#3}
14680     { \msg_error:nne { str } { native-escaping } {#3} }
14681     #1
14682     \else:
14683     #4 #2 #1
14684     \fi:
14685 }

```

(End of definition for __str_convert:wwwnn and __str_convert:NNnNN.)

__str_convert:nnn The arguments of __str_convert:nnn are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to __str_convert:nnnn. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_tmp_tl` tells us what file to load. Loading is skipped if the file was already read, i.e., if the conversion command based on `\l__str_tmp_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_tmp_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_tmp_tl`-based function: we mustn't clobber that different definition.

```

14686 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
14687 {
14688     \cs_if_exist:cF { \__str_convert_#2_#3: }
14689     {
14690         \exp_args:Ne \__str_convert:nnnn
14691         { \__str_convert_lowercase_alphanum:n {#3} }
14692         {#1} {#2} {#3}
14693     }
14694 }
14695 \cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
14696 {
14697     \cs_if_exist:cF { \__str_convert_#3_#1: }
14698     {
14699         \prop_get:NnNF \g__str_alias_prop {#1} \l__str_tmp_tl
14700         { \tl_set:Nn \l__str_tmp_tl {#1} }

```

```

14701     \cs_if_exist:cF { __str_convert_#3_ \l__str_tmp_tl : }
14702     {
14703         \file_if_exist:nTF { l3str-#2- \l__str_tmp_tl .def }
14704         {
14705             \group_begin:
14706             \cctab_select:N \c_code_cctab
14707             \file_input:n { l3str-#2- \l__str_tmp_tl .def }
14708             \group_end:
14709         }
14710         {
14711             \tl_clear:N \l__str_tmp_tl
14712             \msg_error:nnee { str } { unknown-#2 } {#4} {#1}
14713         }
14714     }
14715     \cs_if_exist:cF { __str_convert_#3_#1: }
14716     {
14717         \cs_gset_eq:cc { __str_convert_#3_#1: }
14718         { __str_convert_#3_ \l__str_tmp_tl : }
14719     }
14720 }
14721 \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
14722 }

```

(End of definition for __str_convert:nnn and __str_convert:nnnn.)

__str_convert_lowercase_alphanum:n
__str_convert_lowercase_alphanum_loop:N

This function keeps only letters and digits, with upper case letters converted to lower case.

```

14723 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
14724 {
14725     \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
14726     \tl_to_str:n {#1} { ? \prg_break: }
14727     \prg_break_point:
14728 }
14729 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
14730 {
14731     \use_none:n #1
14732     \if_int_compare:w '#1 > 'Z \exp_stop_f:
14733     \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
14734         \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
14735             #1
14736         \fi:
14737     \fi:
14738     \else:
14739         \if_int_compare:w '#1 < 'A \exp_stop_f:
14740         \if_int_compare:w 1 < 1#1 \exp_stop_f:
14741             #1
14742         \fi:
14743     \else:
14744         \__str_output_byte:n { '#1 + 'a - 'A }
14745     \fi:
14746     \fi:
14747     \__str_convert_lowercase_alphanum_loop:N
14748 }

```

(End of definition for `\_str_convert_lowercase_alphanum:n` and `\_str_convert_lowercase_alphanum-loop:N`.)

60.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `\l__str_byte_flag`. Spaces have already been given the correct category code when this function is called.

`\_str_filter_bytes:n`
`\_str_filter_bytes_aux:N`

```

14749 \bool_lazy_any:nTF
14750 {
14751   \sys_if_engine luatex_p:
14752   \sys_if_engine xetex_p:
14753 }
14754 {
14755   \cs_new:Npn \_str_filter_bytes:n #1
14756   {
14757     \_str_filter_bytes_aux:N #1
14758     { ? \prg_break: }
14759     \prg_break_point:
14760   }
14761   \cs_new:Npn \_str_filter_bytes_aux:N #1
14762   {
14763     \use_none:n #1
14764     \if_int_compare:w '#1 < 256 \exp_stop_f:
14765     #1
14766     \else:
14767       \flag_raise:N \l__str_byte_flag
14768     \fi:
14769     \_str_filter_bytes_aux:N
14770   }
14771 }
14772 { \cs_new_eq:NN \_str_filter_bytes:n \use:n }
```

(End of definition for `\_str_filter_bytes:n` and `\_str_filter_bytes_aux:N`.)

`\_str_convert_unescape_:` The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

`\_str_convert_unescape_bytes:`

```

14773 \bool_lazy_any:nTF
14774 {
14775   \sys_if_engine luatex_p:
14776   \sys_if_engine xetex_p:
14777 }
14778 {
14779   \cs_new_protected:Npn \_str_convert_unescape_:
14780   {
14781     \flag_clear:N \l__str_byte_flag
14782     \__kernel_tl_gset:Nx \g__str_result_tl
14783     { \exp_args:No \_str_filter_bytes:n \g__str_result_tl }
14784     \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { bytes }
14785   }
14786 }
```

```

14787 { \cs_new_protected:Npn \__str_convert_unescape_: { } }
14788 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:

```

(End of definition for __str_convert_unescape_: and __str_convert_unescape_bytes:.)

__str_convert_escape_: The simplest form of escape leaves the bytes from the previous step of the conversion
 __str_convert_escape_bytes: unchanged.

```

14789 \cs_new_protected:Npn \__str_convert_escape_: { }
14790 \cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:

```

(End of definition for __str_convert_escape_: and __str_convert_escape_bytes:.)

60.3.6 Native strings

__str_convert_decode_: Convert each character to its character code, one at a time.
 __str_decode_native_char:N

```

14791 \cs_new_protected:Npn \__str_convert_decode_:
14792 { \__str_convert_gmap:N \__str_decode_native_char:N }
14793 \cs_new:Npn \__str_decode_native_char:N #1
14794 { #1 \s__str \int_value:w '#1 \s__str }

```

(End of definition for __str_convert_decode_: and __str_decode_native_char:N.)

__str_convert_encode_: The conversion from an internal string to native character tokens basically maps \char_-
 __str_encode_native_char:n generate:nn through the code-points, but in non-Unicode-aware engines we use a fall-
 back character ? rather than nothing when given a character code outside [0, 255]. We
 detect the presence of bad characters using a flag and only produce a single error after
 the e-expanding assignment.

```

14795 \bool_lazy_any:nTF
14796 {
14797   \sys_if_engine luatex_p:
14798   \sys_if_engine xetex_p:
14799 }
14800 {
14801   \cs_new_protected:Npn \__str_convert_encode_:
14802   { \__str_convert_gmap_internal:N \__str_encode_native_char:n }
14803   \cs_new:Npn \__str_encode_native_char:n #1
14804   { \char_generate:nn {#1} {12} }
14805 }
14806 {
14807   \cs_new_protected:Npn \__str_convert_encode_:
14808   {
14809     \flag_clear:N \l__str_error_flag
14810     \__str_convert_gmap_internal:N \__str_encode_native_char:n
14811     \__str_if_flag_error:Nne \l__str_error_flag
14812     { native-overflow } { }
14813   }
14814   \cs_new:Npn \__str_encode_native_char:n #1
14815   {
14816     \if_int_compare:w #1 > \c__str_max_byte_int
14817     \flag_raise:N \l__str_error_flag
14818     ?
14819     \else:
14820     \char_generate:nn {#1} {12}
14821     \fi:

```

```

14822     }
14823     \msg_new:nnnn { str } { native-overflow }
14824     { Character-code-too-large-for-this-engine. }
14825     {
14826         This-engine-only-support-8-bit-characters:~
14827         valid-character-codes-are-in-the-range-[0,255].~
14828         To-manipulate-arbitrary-Unicode,~use~LuaTeX-or~XeTeX.
14829     }
14830 }

```

(End of definition for `\__str_convert_encode_:` and `\__str_encode_native_char:n`.)

60.3.7 `clist`

`\__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a `clist` variable, as this avoids problems with leading or trailing commas.

```

14831 \cs_new_protected:Npn \__str_convert_decode_clist:
14832 {
14833     \clist_gset:No \g__str_result_tl \g__str_result_tl
14834     \__kernel_tl_gset:Nx \g__str_result_tl
14835     {
14836         \exp_args:No \clist_map_function:nN
14837         \g__str_result_tl \__str_decode_clist_char:n
14838     }
14839 }
14840 \cs_new:Npn \__str_decode_clist_char:n #1
14841 { #1 \s__str \int_eval:n {#1} \s__str }

```

(End of definition for `\__str_convert_decode_clist:` and `\__str_decode_clist_char:n`.)

`\__str_convert_encode_clist:` Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```

14842 \cs_new_protected:Npn \__str_convert_encode_clist:
14843 {
14844     \__str_convert_gmap_internal:N \__str_encode_clist_char:n
14845     \__kernel_tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
14846 }
14847 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

```

(End of definition for `\__str_convert_encode_clist:` and `\__str_encode_clist_char:n`.)

60.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

```

\__str_declare_eight_bit_encoding:nnnn {<name>} {<modulo>} {<mapping>}
{<missing>}

```


This declares the encoding $\langle name \rangle$ to map bytes to Unicode characters according to the $\langle mapping \rangle$, and map those bytes which are not mentioned in the $\langle mapping \rangle$ either to the replacement character (if they appear in $\langle missing \rangle$), or to themselves. The $\langle mapping \rangle$ argument is a token list of pairs $\{\langle byte \rangle\} \{\langle Unicode \rangle\}$ expressed in uppercase hexadecimal notation. The $\langle missing \rangle$ argument is a token list of $\{\langle byte \rangle\}$. Every $\langle byte \rangle$ which does not appear in the $\langle mapping \rangle$ nor the $\langle missing \rangle$ lists maps to itself in Unicode, so for instance the `latin1` encoding has empty $\langle mapping \rangle$ and $\langle missing \rangle$ lists. The $\langle modulo \rangle$ is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `decode` integer array, with entry $n + 1$ (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the n -th byte in the encoding under consideration, or -1 if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point n , we look up the entry $(1 \text{ plus } n \text{ modulo some number } M)$ in the `encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `decode` array that indeed this byte encodes the Unicode code point we want. Here, M is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo M .

Loop through both lists of bytes to fill in the `decode` integer array, then fill the `encode` array accordingly. For bytes that are invalid in the given encoding, store -1 in the `decode` array.

```

\__str_declare_eight_bit_encoding:nnnn
\__str_declare_eight_bit_aux:NNnnn
\__str_declare_eight_bit_loop:Nnn
\__str_declare_eight_bit_loop:Nn
14848 \cs_new_protected:Npn \__str_declare_eight_bit_encoding:nnnn #1
14849 {
14850   \tl_set:Nn \l__str_tmp_tl {#1}
14851   \cs_new_protected:cpn { __str_convert_decode_#1: }
14852     { \__str_convert_decode_eight_bit:n {#1} }
14853   \cs_new_protected:cpn { __str_convert_encode_#1: }
14854     { \__str_convert_encode_eight_bit:n {#1} }
14855   \exp_args:Ncc \__str_declare_eight_bit_aux:NNnnn
14856     { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
14857 }
14858 \cs_new_protected:Npn \__str_declare_eight_bit_aux:NNnnn #1#2#3#4#5
14859 {
14860   \intarray_new:Nn #1 { 256 }
14861   \int_step_inline:nnn { 0 } { 255 }
14862     { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
14863   \__str_declare_eight_bit_loop:Nnn #1
14864     #4 { \s__str_stop \prg_break: } { }
14865   \prg_break_point:
14866   \__str_declare_eight_bit_loop:Nn #1
14867     #5 { \s__str_stop \prg_break: }
14868   \prg_break_point:
14869   \intarray_new:Nn #2 {#3}
14870   \int_step_inline:nnn { 0 } { 255 }
14871   {
14872     \int_compare:nNf { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
14873     {
14874       \intarray_gset:Nnn #2
14875         {
14876           1 +

```

```

14877         \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
14878         { \intarray_count:N #2 }
14879     }
14880     {##1}
14881 }
14882 }
14883 }
14884 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
14885 {
14886     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14887     \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
14888     \__str_declare_eight_bit_loop:Nnn #1
14889 }
14890 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
14891 {
14892     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14893     \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
14894     \__str_declare_eight_bit_loop:Nn #1
14895 }

```

(End of definition for __str_declare_eight_bit_encoding:nnnn and others.)

```

\__str_convert_decode_eight_bit:n
  \__str_decode_eight_bit_aux:n
  \__str_decode_eight_bit_aux:Nn

```

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `\__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is `-1` then issue a replacement character and raise the flag indicating that there was an error.

```

14896 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
14897 {
14898     \cs_set:Npe \__str_tmp:w
14899     {
14900         \exp_not:N \__str_decode_eight_bit_aux:Nn
14901         \exp_not:c { g__str_decode_#1_intarray }
14902     }
14903     \flag_clear:N \l__str_error_flag
14904     \__str_convert_gmap:N \__str_tmp:w
14905     \__str_if_flag_error:Nne \l__str_error_flag { decode-8-bit } {#1}
14906 }
14907 \cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
14908 {
14909     #2 \s__str
14910     \exp_args:Nf \__str_decode_eight_bit_aux:n
14911     { \intarray_item:Nn #1 { 1 + '#2 } }
14912     \s__str
14913 }
14914 \cs_new:Npn \__str_decode_eight_bit_aux:n #1
14915 {
14916     \if_int_compare:w #1 < \c_zero_int
14917         \flag_raise:N \l__str_error_flag
14918         \int_value:w \c__str_replacement_char_int
14919     \else:
14920         #1
14921     \fi:
14922 }

```

(End of definition for `__str_convert_encodeEightBit:n`, `__str_decodeEightBit_aux:n`, and `__str_decodeEightBit_aux:Nn`.)

```
__str_convert_encodeEightBit:n
__str_encodeEightBit_aux:nnN
__str_encodeEightBit_aux:NnN
```

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```
14923 \int_new:N \l__str_modulo_int
14924 \cs_new_protected:Npn \__str_convert_encodeEightBit:n #1
14925 {
14926   \cs_set:Npe \__str_tmp:w
14927   {
14928     \exp_not:N \__str_encodeEightBit_aux:NnN
14929     \exp_not:c { g__str_encode_#1_intarray }
14930     \exp_not:c { g__str_decode_#1_intarray }
14931   }
14932   \flag_clear:N \l__str_error_flag
14933   \__str_convert_gmap_internal:N \__str_tmp:w
14934   \__str_if_flag_error:Nne \l__str_error_flag { encode-8-bit } {#1}
14935 }
14936 \cs_new:Npn \__str_encodeEightBit_aux:nnN #1#2#3
14937 {
14938   \exp_args:Nf \__str_encodeEightBit_aux:nnN
14939   {
14940     \intarray_item:Nn #1
14941     { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }
14942   }
14943   {#3}
14944   #2
14945 }
14946 \cs_new:Npn \__str_encodeEightBit_aux:nnN #1#2#3
14947 {
14948   \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
14949   { \__str_output_byte:n {#1} }
14950   { \flag_raise:N \l__str_error_flag }
14951 }
```

(End of definition for `__str_convert_encodeEightBit:n`, `__str_encodeEightBit_aux:nnN`, and `__str_encodeEightBit_aux:NnN`.)

60.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```
14952 \msg_new:nnn { str } { unknown-esc }
14953 { Escaping-scheme~'#1'~(filtered:~'#2')~unknown. }
14954 \msg_new:nnn { str } { unknown-enc }
14955 { Encoding-scheme~'#1'~(filtered:~'#2')~unknown. }
14956 \msg_new:nnnn { str } { native-escaping }
14957 { The~'native'~encoding~scheme~does~not~support~any~escaping. }
14958 {
```

```

14959     Since~native~strings~do~not~consist~in~bytes,~
14960     none~of~the~escaping~methods~make~sense.~
14961     The~specified~escaping,~'#1',~will~be~ignored.
14962   }
14963   \msg_new:nnn { str } { file-not-found }
14964   { File~'l3str-#1.def'~not~found. }

```

Message used when the “bytes” unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```

14965   \bool_lazy_any:nT
14966   {
14967     \sys_if_engine luatex_p:
14968     \sys_if_engine xetex_p:
14969   }
14970   {
14971     \msg_new:nnnn { str } { non-byte }
14972     { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
14973     {
14974       Some~characters~in~the~string~you~asked~to~convert~are~not~
14975       8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
14976       If~it~is,~try~using\\
14977       \\
14978       \iow_indent:n
14979       {
14980         \iow_char:N\\str_set_convert:Nnnn \\
14981         \ \ <str-var>~\{~<string>~\}~\{~<native>~\}~\{~<target-encoding>~\}
14982       }
14983     }
14984   }

```

Those messages are used when converting to and from 8-bit encodings.

```

14985   \msg_new:nnnn { str } { decode-8-bit }
14986   { Invalid-string-in-encoding~'#1'. }
14987   {
14988     LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
14989     any~character~in~the~encoding~'#1'.
14990   }
14991   \msg_new:nnnn { str } { encode-8-bit }
14992   { Unicode-string~cannot~be~converted~to~encoding~'#1'. }
14993   {
14994     The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
14995     LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
14996     string~contains~a~character~that~'#1'~does~not~support.
14997   }

```

60.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);

- `hex` or `hexadecimal`, as per the pdfTeX primitive `\pdfescapehex`
- `name`, as per the pdfTeX primitive `\pdfescapename`
- `string`, as per the pdfTeX primitive `\pdfescapestring`
- `url`, as per the percent encoding of urls.

60.5.1 Unescape methods

`\__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
`\__str_unescape_hex_auxi:N` Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
`\__str_unescape_hex_auxii:N` an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```

14998 \cs_new_protected:Npn \__str_convert_unescape_hex:
14999 {
15000   \group_begin:
15001     \flag_clear:N \l__str_error_flag
15002     \int_set:Nn \tex_escapechar:D { 92 }
15003     \__kernel_tl_gset:Nx \g__str_result_tl
15004     {
15005       \__str_output_byte:w "
15006       \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
15007       { \tl_to_str:N \g__str_result_tl }
15008       0 { ? 0 - 1 \prg_break: }
15009       \prg_break_point:
15010       \__str_output_end:
15011     }
15012     \__str_if_flag_error:Nne \l__str_error_flag { unescape-hex } { }
15013   \group_end:
15014 }
15015 \cs_new:Npn \__str_unescape_hex_auxi:N #1
15016 {
15017   \use_none:n #1
15018   \__str_hexadecimal_use:NTF #1
15019   { \__str_unescape_hex_auxii:N }
15020   {
15021     \flag_raise:N \l__str_error_flag
15022     \__str_unescape_hex_auxi:N
15023   }
15024 }
15025 \cs_new:Npn \__str_unescape_hex_auxii:N #1
15026 {
15027   \use_none:n #1
15028   \__str_hexadecimal_use:NTF #1
15029   {
15030     \__str_output_end:
15031     \__str_output_byte:w " \__str_unescape_hex_auxi:N
15032   }
15033   {
15034     \flag_raise:N \l__str_error_flag
15035     \__str_unescape_hex_auxii:N
15036   }
15037 }

```

```

15038 \msg_new:nnnn { str } { unescape-hex }
15039 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
15040 {
15041   Some~characters~in~the~string~you~asked~to~convert~are~not~
15042   hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
15043 }

```

(End of definition for `\__str_convert_unescape_hex:`, `\__str_unescape_hex_auxi:N`, and `\__str_unescape_hex_auxii:N`.)

```

\__str_convert_unescape_name:
\__str_unescape_name_loop:wNN
\__str_convert_unescape_url:
\__str_unescape_url_loop:wNN

```

The `\__str_convert_unescape_name:` function replaces each occurrence of # followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character % instead of #. Thus we define the two together. The arguments of `\__str_tmp:w` are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test `\__str_hexadecimal_use:N` leaves the upper-case digit in the input stream, hence we surround the test with `\__str_output_byte:w` and `\__str_output_end:.` If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed "#1 to `\__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```

15044 \cs_set_protected:Npn \__str_tmp:w #1#2#3
15045 {
15046   \cs_new_protected:cpn { __str_convert_unescape_#2: }
15047   {
15048     \group_begin:
15049     \flag_clear:N \l__str_byte_flag
15050     \flag_clear:N \l__str_error_flag
15051     \int_set:Nn \tex_escapechar:D { 92 }
15052     \__kernel_tl_gset:Nx \g__str_result_tl
15053     {
15054       \exp_after:wN #3 \g__str_result_tl
15055       #1 ? { ? \prg_break: }
15056       \prg_break_point:
15057     }
15058     \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { #2 }
15059     \__str_if_flag_error:Nne \l__str_error_flag { unescape-#2 } { }
15060   \group_end:
15061 }
15062 \cs_new:Npn #3 ##1#1##2##3
15063 {
15064   \__str_filter_bytes:n {##1}
15065   \use_none:n ##3
15066   \__str_output_byte:w "
15067   \__str_hexadecimal_use:NTF ##2
15068   {
15069     \__str_hexadecimal_use:NTF ##3
15070     { }
15071     {
15072       \flag_raise:N \l__str_error_flag

```

```

15073         * 0 + '#1 \use_i:nn
15074     }
15075 }
15076 {
15077     \flag_raise:N \l__str_error_flag
15078     0 + '#1 \use_i:nn
15079 }
15080 \__str_output_end:
15081 \use_i:nnn #3 ##2##3
15082 }
15083 \msg_new:nnnn { str } { unescape-#2 }
15084 { String~invalid~in~escaping~'#2'. }
15085 {
15086     LaTeX~came~across~the~escape~character~'~#1'~not~followed~by~
15087     two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'~#2'.
15088 }
15089 }
15090 \exp_after:wN \__str_tmp:w \c_hash_str { name }
15091 \__str_unescape_name_loop:wNN
15092 \exp_after:wN \__str_tmp:w \c_percent_str { url }
15093 \__str_unescape_url_loop:wNN

```

(End of definition for `\__str_convert_unescape_name:` and others.)

```

\__str_convert_unescape_string:
\__str_unescape_string_newlines:wN
\__str_unescape_string_loop:wNNN
\__str_unescape_string_repeat:NNNNN

```

The **string** escaping is somewhat similar to the **name** and **url** escapings, with escape character `\`. The first step is to convert all three line endings, `^^J`, `^^M`, and `^^M^^J` to the common `^^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

```

\~n Line feed (10)
\~r Carriage return (13)
\~t Horizontal tab (9)
\~b Backspace (8)
\~f Form feed (12)
\~( Left parenthesis
\~) Right parenthesis
\~\ Backslash

```

`\ddd` (backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```

15094 \group_begin:
15095 \char_set_catcode_other:N \^^J
15096 \char_set_catcode_other:N \^^M
15097 \cs_set_protected:Npn \__str_tmp:w #1
15098 {
15099     \cs_new_protected:Npn \__str_convert_unescape_string:

```

```

15100 {
15101   \group_begin:
15102     \flag_clear:N \l__str_byte_flag
15103     \flag_clear:N \l__str_error_flag
15104     \int_set:Nn \tex_escapechar:D { 92 }
15105     \__kernel_tl_gset:Nx \g__str_result_tl
15106       {
15107         \exp_after:wN \__str_unescape_string_newlines:wN
15108         \g__str_result_tl \prg_break: ^M ?
15109         \prg_break_point:
15110       }
15111     \__kernel_tl_gset:Nx \g__str_result_tl
15112       {
15113         \exp_after:wN \__str_unescape_string_loop:wNNN
15114         \g__str_result_tl #1 ?? { ? \prg_break: }
15115         \prg_break_point:
15116       }
15117     \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { string }
15118     \__str_if_flag_error:Nne \l__str_error_flag { unescape-string } { }
15119   \group_end:
15120 }
15121 }
15122 \exp_args:No \__str_tmp:w { \c_backslash_str }
15123 \exp_last_unbraced:NNNNo
15124 \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
15125 {
15126   \__str_filter_bytes:n {#1}
15127   \use_none:n #4
15128   \__str_output_byte:w '
15129   \__str_octal_use:NTF #2
15130   {
15131     \__str_octal_use:NTF #3
15132     {
15133       \__str_octal_use:NTF #4
15134       {
15135         \if_int_compare:w #2 > 3 \exp_stop_f:
15136         - 256
15137         \fi:
15138         \__str_unescape_string_repeat:NNNNNN
15139       }
15140       { \__str_unescape_string_repeat:NNNNNN ? }
15141     }
15142     { \__str_unescape_string_repeat:NNNNNN ?? }
15143   }
15144   {
15145     \str_case_e:nnF {#2}
15146     {
15147       { \c_backslash_str } { 134 }
15148       { ( ) } { 50 }
15149       { ) } { 51 }
15150       { r } { 15 }
15151       { f } { 14 }
15152       { n } { 12 }
15153       { t } { 11 }

```



```

15154         { b } { 10 }
15155         { ^^J } { 0 - 1 }
15156     }
15157     {
15158         \flag_raise:N \l__str_error_flag
15159         0 - 1 \use_i:nn
15160     }
15161 }
15162 \__str_output_end:
15163 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
15164 }
15165 \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
15166 { \__str_output_end: \__str_unescape_string_loop:wNNN }
15167 \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
15168 {
15169     #1
15170     \if_charcode:w ^^J #2 \else: ^^J \fi:
15171     \__str_unescape_string_newlines:wN #2
15172 }
15173 \msg_new:nnnn { str } { unescape-string }
15174 { String~invalid~in~escaping~'string'. }
15175 {
15176     LaTeX~came~across~an~escape~character~'\c_backslash_str'~
15177     not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(' ,~')',~
15178     '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
15179     of~a~line.
15180 }
15181 \group_end:

```

(End of definition for `\__str_convert_unescape_string:` and others.)

60.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

```

\__str_convert_escape_hex: Loop and convert each byte to hexadecimal.
\__str_escape_hex_char:N
15182 \cs_new_protected:Npn \__str_convert_escape_hex:
15183 { \__str_convert_gmap:N \__str_escape_hex_char:N }
15184 \cs_new:Npn \__str_escape_hex_char:N #1
15185 { \__str_output_hexadecimal:n { '#1' } }

(End of definition for \__str_convert_escape_hex: and \__str_escape_hex_char:N.)

\__str_convert_escape_name: For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly,
\__str_escape_name_char:n bytes outside the range ["2A,"7E] are hash-encoded. We keep two lists of exceptions:
\__str_if_escape_name:nTF characters in \c__str_escape_name_not_str are not hash-encoded, and characters in
\c__str_escape_name_str the \c__str_escape_name_str are encoded.
\c__str_escape_name_not_str
15186 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
15187 \str_const:Nn \c__str_escape_name_str { { } / < > [ ] }
15188 \cs_new_protected:Npn \__str_convert_escape_name:
15189 { \__str_convert_gmap:N \__str_escape_name_char:n }
15190 \cs_new:Npn \__str_escape_name_char:n #1
15191 {

```

```

15192     \__str_if_escape_name:nTF {#1} {#1}
15193     { \c_hash_str \__str_output_hexadecimal:n {'#1} }
15194 }
15195 \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
15196 {
15197     \if_int_compare:w '#1 < "2A \exp_stop_f:
15198     \__str_if_contains_char:NnTF \c__str_escape_name_not_str {#1}
15199     \prg_return_true: \prg_return_false:
15200 }else:
15201     \if_int_compare:w '#1 > "7E \exp_stop_f:
15202     \prg_return_false:
15203 }else:
15204     \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
15205     \prg_return_false: \prg_return_true:
15206 }fi:
15207 \fi:
15208 }

```

(End of definition for __str_convert_escape_name: and others.)

__str_convert_escape_string: Any character below (and including) space, and any character above (and including) del, are converted to octal. One backslash is added before each parenthesis and backslash.

```

\__str_escape_string_char:N
\__str_if_escape_string:N
\c__str_escape_string_str
15209 \str_const:Ne \c__str_escape_string_str
15210 { \c_backslash_str ( ) }
15211 \cs_new_protected:Npn \__str_convert_escape_string:
15212 { \__str_convert_gmap:N \__str_escape_string_char:N }
15213 \cs_new:Npn \__str_escape_string_char:N #1
15214 {
15215     \__str_if_escape_string:NTF #1
15216     {
15217         \__str_if_contains_char:NnT
15218         \c__str_escape_string_str {#1}
15219         { \c_backslash_str }
15220         #1
15221     }
15222     {
15223         \c_backslash_str
15224         \int_div_truncate:nn {'#1} {64}
15225         \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
15226         \int_mod:nn {'#1} { 8 }
15227     }
15228 }
15229 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
15230 {
15231     \if_int_compare:w '#1 < "27 \exp_stop_f:
15232     \prg_return_false:
15233 }else:
15234     \if_int_compare:w '#1 > "7A \exp_stop_f:
15235     \prg_return_false:
15236 }else:
15237     \prg_return_true:
15238 }fi:
15239 \fi:
15240 }

```

(End of definition for `__str_convert_escape_string:` and others.)

`__str_convert_escape_url:` This function is similar to `__str_convert_escape_name:`, escaping different characters.

```

__str_escape_url_char:n
__str_if_escape_url:nTF
15241 \cs_new_protected:Npn __str_convert_escape_url:
15242 { __str_convert_gmap:N __str_escape_url_char:n }
15243 \cs_new:Npn __str_escape_url_char:n #1
15244 {
15245   __str_if_escape_url:nTF {#1} {#1}
15246   { \c_percent_str __str_output_hexadecimal:n { '#1' } }
15247 }
15248 \prg_new_conditional:Npnn __str_if_escape_url:n #1 { TF }
15249 {
15250   \if_int_compare:w '#1 < "30 \exp_stop_f:
15251   __str_if_contains_char:nnTF { "-. } {#1}
15252   \prg_return_true: \prg_return_false:
15253   \else:
15254   \if_int_compare:w '#1 > "7E \exp_stop_f:
15255   \prg_return_false:
15256   \else:
15257   __str_if_contains_char:nnTF { : ; = ? @ [ ] } {#1}
15258   \prg_return_false: \prg_return_true:
15259   \fi:
15260   \fi:
15261 }

```

(End of definition for `__str_convert_escape_url:`, `__str_escape_url_char:n`, and `__str_if_escape_url:nTF`.)

60.6 Encoding definitions

The **native** encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

60.6.1 utf-8 support

```

__str_convert_encode_utf8:
  __str_encode_utf_viii_char:n
  __str_encode_utf_viii_loop:wwnnw

```

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the

residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0,15], which we output shifted by 224. The last range, [65536,1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `\__str_encode_utf_vii_loop:wnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0,127], [192,223], [224,239], and [240,247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient `#1` is less than the limit `#3` for that range, output the leading byte (`#1` shifted by `#4`) and stop. Otherwise, we need one more step: use the quotient of `#1` by 64, and `#1` as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder `#2 - 64#1 + 128`. The bizarre construction `- 1 + 0 *` removes the spurious initial continuation byte (better methods welcome).

```

15262 \cs_new_protected:cpn { __str_convert_encode_utf8: }
15263 { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
15264 \cs_new:Npn \__str_encode_utf_viii_char:n #1
15265 {
15266   \__str_encode_utf_viii_loop:wnnw #1 \__str_sep: - 1 + 0 * \__str_sep:
15267   { 128 } { 0 }
15268   { 32 } { 192 }
15269   { 16 } { 224 }
15270   { 8 } { 240 }
15271   \s__str_stop
15272 }
15273 \cs_new:Npn \__str_encode_utf_viii_loop:wnnw
15274 #1 \__str_sep: #2 \__str_sep: #3#4 #5 \s__str_stop
15275 {
15276   \if_int_compare:w #1 < #3 \exp_stop_f:
15277     \__str_output_byte:n { #1 + #4 }
15278     \exp_after:wN \__str_use_none_delimit_by_s_stop:w
15279   \fi:
15280   \exp_after:wN \__str_encode_utf_viii_loop:wnnw
15281     \int_value:w \int_div_truncate:nn {#1} {64} \__str_sep: #1 \__str_sep:
15282     #5 \s__str_stop
15283   \__str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
15284 }

```

(End of definition for `\__str_convert_encode_utf8:`, `\__str_encode_utf_viii_char:n`, and `\__str_encode_utf_viii_loop:wnnw`.)

<code>__str_missing</code> <code>__str_extra</code> <code>__str_overlong</code> <code>__str_overflow</code>	When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signaled by a specific flag for each (we define those flags using <code>\flag_clear_new:N</code> rather than <code>\flag_new:N</code> , because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, i.e., not after an appropriate leading byte.

- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

15285 \flag_clear_new:N \l__str_missing_flag
15286 \flag_clear_new:N \l__str_extra_flag
15287 \flag_clear_new:N \l__str_overlong_flag
15288 \flag_clear_new:N \l__str_overflow_flag
15289 \msg_new:nnnn { str } { utf8-decode }
15290 {
15291     Invalid-UTF-8-string:
15292     \exp_last_unbraced:Nf \use_none:n
15293     {
15294         \__str_if_flag_times:NT \l__str_missing_flag { ,~missing~continuation~byte }
15295         \__str_if_flag_times:NT \l__str_extra_flag { ,~extra~continuation~byte }
15296         \__str_if_flag_times:NT \l__str_overlong_flag { ,~overlong~form }
15297         \__str_if_flag_times:NT \l__str_overflow_flag { ,~code~point~too~large }
15298     }
15299 }
15300 }
15301 {
15302     In-the-UTF-8-encoding,~each-Unicode-character-consists-in-
15303     1~to~4~bytes,~with-the-following-bit-pattern: \
15304     \iow_indent:n
15305     {
15306         Code-point~\ \ \ \ <~128:~0xxxxxxx \
15307         Code-point~\ \ \ \ <~2048:~110xxxxx~10xxxxxx \
15308         Code-point~\ \ \ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \
15309         Code-point~ \ \ \ \ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \
15310     }
15311     Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.
15312     \flag_if_raised:NT \l__str_missing_flag
15313     {
15314         \\\
15315         A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
15316         the~appropriate~number~of~continuation~bytes.
15317     }
15318     \flag_if_raised:NT \l__str_extra_flag
15319     {
15320         \\\
15321         LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
15322     }
15323     \flag_if_raised:NT \l__str_overlong_flag
15324     {
15325         \\\
15326         Every~Unicode~code~point~must~be~expressed~in~the~shortest~
15327         possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
15328         representation~for~the~code~point~3.

```

```

15329     }
15330     \flag_if_raised:NT \l__str_overflow_flag
15331     {
15332         \\\
15333         Unicode~limits~code~points~to~the~range~[0,1114111] .
15334     }
15335 }
15336 \prop_gput:Nnn \g_msg_module_name_prop { str } { LaTeX }
15337 \prop_gput:Nnn \g_msg_module_type_prop { str } { }

```

(End of definition for `__str_missing` and others.)

`__str_convert_decode_utf8:` Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX3 error, as explained above). We expect successive multi-byte sequences of the form *⟨start byte⟩ ⟨continuation bytes⟩*. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to –"C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

15338 \cs_new_protected:cpn { __str_convert_decode_utf8: }
15339 {

```

```

15340 \flag_clear:N \l__str_error_flag
15341 \flag_clear:N \l__str_missing_flag
15342 \flag_clear:N \l__str_extra_flag
15343 \flag_clear:N \l__str_overlong_flag
15344 \flag_clear:N \l__str_overflow_flag
15345 \__kernel_tl_gset:Nx \g__str_result_tl
15346 {
15347   \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
15348   { \prg_break: \__str_decode_utf_viii_end: }
15349   \prg_break_point:
15350 }
15351 \__str_if_flag_error:Nne \l__str_error_flag { utf8-decode } { }
15352 }
15353 \cs_new:Npn \__str_decode_utf_viii_start:N #1
15354 {
15355   #1
15356   \if_int_compare:w '#1 < "C0 \exp_stop_f:
15357     \s__str
15358     \if_int_compare:w '#1 < "80 \exp_stop_f:
15359       \int_value:w '#1
15360     \else:
15361       \flag_raise:N \l__str_extra_flag
15362       \flag_raise:N \l__str_error_flag
15363       \int_use:N \c__str_replacement_char_int
15364     \fi:
15365   \else:
15366     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
15367     \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
15368   \fi:
15369   \s__str
15370   \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
15371   \__str_decode_utf_viii_start:N
15372 }
15373 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
15374   #1 \s__str #2 \__str_decode_utf_viii_start:N #3
15375 {
15376   \use_none:n #3
15377   \if_int_compare:w '#3 <
15378     \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
15379     "C0 \exp_stop_f:
15380   #3
15381   \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
15382   \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
15383   \else:
15384     \s__str
15385     \flag_raise:N \l__str_missing_flag
15386     \flag_raise:N \l__str_error_flag
15387     \int_use:N \c__str_replacement_char_int
15388   \fi:
15389   \s__str
15390   #2
15391   \__str_decode_utf_viii_start:N #3
15392 }
15393 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN

```

```

15394     #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
15395   {
15396     \if_int_compare:w #1 < #4 \exp_stop_f:
15397       \s__str
15398       \if_int_compare:w #1 < #3 \exp_stop_f:
15399         \flag_raise:N \l__str_overlong_flag
15400         \flag_raise:N \l__str_error_flag
15401         \int_use:N \c__str_replacement_char_int
15402       \else:
15403         #1
15404       \fi:
15405     \else:
15406       \if_meaning:w \s__str_stop #5
15407         \__str_decode_utf_viii_overflow:w #1
15408       \fi:
15409       \exp_after:wN \__str_decode_utf_viii_continuation:wwN
15410       \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
15411     \fi:
15412     \s__str
15413     #2 {#4} #5
15414     \__str_decode_utf_viii_start:N
15415   }
15416 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
15417 {
15418   \fi: \fi:
15419   \flag_raise:N \l__str_overflow_flag
15420   \flag_raise:N \l__str_error_flag
15421   \int_use:N \c__str_replacement_char_int
15422 }
15423 \cs_new:Npn \__str_decode_utf_viii_end:
15424 {
15425   \s__str
15426   \flag_raise:N \l__str_missing_flag
15427   \flag_raise:N \l__str_error_flag
15428   \int_use:N \c__str_replacement_char_int \s__str
15429   \prg_break:
15430 }

```

(End of definition for `\__str_convert_decode_utf8:` and others.)

60.6.2 utf-16 support

The definitions are done in a category code régime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

15431 \group_begin:
15432   \char_set_catcode_other:N ^^fe
15433   \char_set_catcode_other:N ^^ff

```

`\__str_convert_encode_utf16:` When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviors depending on the character code.

`\__str_encode_utf_xvi_aux:N`
`\__str_encode_utf_xvi_char:n`

- [0, "D7FF]: converted to two bytes;

- ["D800,"DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000,"FFFF]: converted to two bytes;
- ["10000,"10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 – "10000/"400.

For the duration of this operation, `\__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `\__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

15434 \cs_new_protected:cpn { __str_convert_encode_utf16: }
15435 {
15436   \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
15437   \tl_gput_left:N \g__str_result_tl { ^^fe ^^ff }
15438 }
15439 \cs_new_protected:cpn { __str_convert_encode_utf16be: }
15440 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
15441 \cs_new_protected:cpn { __str_convert_encode_utf16le: }
15442 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
15443 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
15444 {
15445   \flag_clear:N \l__str_error_flag
15446   \cs_set_eq:NN \__str_tmp:w #1
15447   \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
15448   \__str_if_flag_error:Nne \l__str_error_flag { utf16-encode } { }
15449 }
15450 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
15451 {
15452   \if_int_compare:w #1 < "D800 \exp_stop_f:
15453   \__str_tmp:w {#1}
15454   \else:
15455     \if_int_compare:w #1 < "10000 \exp_stop_f:
15456     \if_int_compare:w #1 < "E000 \exp_stop_f:
15457     \flag_raise:N \l__str_error_flag
15458     \__str_tmp:w { \c__str_replacement_char_int }
15459     \else:
15460       \__str_tmp:w {#1}
15461     \fi:
15462     \else:
15463       \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
15464       \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
15465     \fi:
15466   \fi:
15467 }

```

(End of definition for __str_convert_encode_utf16: and others.)

`__str_missing` When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800,"DFFF], corresponding to surrogates, cannot be encoded. We use the `__str_extra` all-purpose flag `@@_error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

15468 \flag_clear_new:N \l__str_missing_flag
15469 \flag_clear_new:N \l__str_extra_flag
15470 \flag_clear_new:N \l__str_end_flag
15471 \msg_new:nnnn { str } { utf16-encode }
15472 { Unicode-string-cannot-be-expressed-in-UTF-16:-surrogate. }
15473 {
15474   Surrogate-code-points~(in-the-range~[U+D800,~U+DFFF])~
15475   can-be-expressed-in-the~UTF-8-and-UTF-32-encodings,~
15476   but-not-in-the~UTF-16-encoding.
15477 }
15478 \msg_new:nnnn { str } { utf16-decode }
15479 {
15480   Invalid-UTF-16-string:
15481   \exp_last_unbraced:Nf \use_none:n
15482   {
15483     \__str_if_flag_times:NT \l__str_missing_flag { ,~missing~trail~surrogate }
15484     \__str_if_flag_times:NT \l__str_extra_flag { ,~extra~trail~surrogate }
15485     \__str_if_flag_times:NT \l__str_end_flag { ,~odd-number~of~bytes }
15486   }
15487   .
15488 }
15489 {
15490   In~the~UTF-16-encoding,~each~Unicode~character~is~encoded~as~
15491   2-or~4-bytes: \\
15492   \iow_indent:n
15493   {
15494     Code-point-in~[U+0000,~U+D7FF]:~two-bytes \\
15495     Code-point-in~[U+D800,~U+DFFF]:~illegal \\
15496     Code-point-in~[U+E000,~U+FFFF]:~two-bytes \\
15497     Code-point-in~[U+10000,~U+10FFFF]:~
15498     a~lead~surrogate~and~a~trail~surrogate \\
15499   }
15500   Lead-surrogates-are-pairs-of-bytes-in-the-range~[0xD800,~0xDBFF],~
15501   and~trail~surrogates-are-in-the-range~[0xDC00,~0xDFFF].
15502   \flag_if_raised:NT \l__str_missing_flag
15503   {
15504     \\ \\
15505     A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
15506   }
15507   \flag_if_raised:NT \l__str_extra_flag
15508   {
15509     \\ \\
15510     LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
15511   }
15512   \flag_if_raised:NT \l__str_end_flag
15513   {
15514     \\ \\
15515     The~string~contained~an~odd-number~of~bytes.~This-is~invalid:~
15516     the~basic~code~unit~for~UTF-16~is~16~bits~(2-bytes).
15517   }
15518 }

```

(End of definition for `__str_missing`, `__str_extra`, and `__str_end`.)

`__str_convert_decode_utf16:` As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_t1` as an argument before expansion is cheaper).

The `__str_decode_utf_xvi:Nw` function defines `__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `__str_decode_utf_xvi_pair:NN` described below.

```

15519 \cs_new_protected:cpn { __str_convert_decode_utf16be: }
15520 { \__str_decode_utf_xvi:Nw 1 \g__str_result_t1 \s__str_stop }
15521 \cs_new_protected:cpn { __str_convert_decode_utf16le: }
15522 { \__str_decode_utf_xvi:Nw 2 \g__str_result_t1 \s__str_stop }
15523 \cs_new_protected:cpn { __str_convert_decode_utf16: }
15524 {
15525     \exp_after:wN \__str_decode_utf_xvi_bom:NN
15526     \g__str_result_t1 \s__str_stop \s__str_stop \s__str_stop
15527 }
15528 \cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
15529 {
15530     \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
15531     { \__str_decode_utf_xvi:Nw 2 }
15532     {
15533         \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
15534         { \__str_decode_utf_xvi:Nw 1 }
15535         { \__str_decode_utf_xvi:Nw 1 #1#2 }
15536     }
15537 }
15538 \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
15539 {
15540     \flag_clear:N \l__str_error_flag
15541     \flag_clear:N \l__str_missing_flag
15542     \flag_clear:N \l__str_extra_flag
15543     \flag_clear:N \l__str_end_flag
15544     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
15545     \__kernel_tl_gset:Nx \g__str_result_t1
15546     {
15547         \exp_after:wN \__str_decode_utf_xvi_pair:NN
15548         #2 \q__str_nil \q__str_nil
15549         \prg_break_point:
15550     }
15551     \__str_if_flag_error:Nne \l__str_error_flag { utf16-decode } { }
15552 }

```

(End of definition for `__str_convert_decode_utf16:` and others.)

`__str_decode_utf_xvi_pair:NN` Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

`__str_decode_utf_xvi_quad:NNwNN`

`__str_decode_utf_xvi_pair_end:Nw`

`__str_decode_utf_xvi_error:nNN`

`__str_decode_utf_xvi_extra:NNw`

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε -TeX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__str <code point> \s__str`, and remove the pair `#4#5` before looping with `\__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7*"400 = "D800*"400+"DC00—"10000.

Every time we read a pair of bytes, we test for the end-marker `\q__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

15553 \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
15554 {
15555   \if_meaning:w \q__str_nil #2
15556   \__str_decode_utf_xvi_pair_end:Nw #1
15557   \fi:
15558   \if_case:w
15559     \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
15560   \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
15561   \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
15562   \fi:
15563   #1#2 \s__str
15564   \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
15565   \__str_decode_utf_xvi_pair:NN
15566 }
15567 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
15568   #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
15569 {
15570   \if_meaning:w \q__str_nil #5
15571   \__str_decode_utf_xvi_error:nNN { missing } #1#2
15572   \__str_decode_utf_xvi_pair_end:Nw #4
15573   \fi:
15574   \if_int_compare:w
15575     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
15576     0 = 1
15577   \else:
15578     \__str_tmp:w #4#5 < "E0
15579   \fi:
15580   \exp_stop_f:
15581   #1 #2 #4 #5 \s__str
15582   \int_eval:n
15583   {

```

```

15584      ( "100 * \_str_tmp:w #1#2 + \_str_tmp:w #2#1 - "D7F7 ) * "400
15585      + "100 * \_str_tmp:w #4#5 + \_str_tmp:w #5#4
15586    }
15587    \s__str
15588    \exp_after:wN \use_i:nnn
15589  \else:
15590    \_str_decode_utf_xvi_error:nNN { missing } #1#2
15591  \fi:
15592  \_str_decode_utf_xvi_pair:NN #4#5
15593  }
15594  \cs_new:Npn \_str_decode_utf_xvi_pair_end:Nw #1 \fi:
15595  {
15596    \fi:
15597    \if_meaning:w \q__str_nil #1
15598  \else:
15599    \_str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
15600  \fi:
15601  \prg_break:
15602  }
15603  \cs_new:Npn \_str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
15604  { \_str_decode_utf_xvi_error:nNN { extra } #1#2 }
15605  \cs_new:Npn \_str_decode_utf_xvi_error:nNN #1#2#3
15606  {
15607    \flag_raise:N \l__str_error_flag
15608    \flag_raise:c { l__str_#1_flag }
15609    #2 #3 \s__str
15610    \int_use:N \c__str_replacement_char_int \s__str
15611  }

```

(End of definition for `\_str_decode_utf_xvi_pair:NN` and others.)

Restore the original catcodes of bytes 254 and 255.

```

15612 \group_end:

```

60.6.3 utf-32 support

The definitions are done in a category code régime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

15613 \group_begin:
15614   \char_set_catcode_other:N ^^00
15615   \char_set_catcode_other:N ^^fe
15616   \char_set_catcode_other:N ^^ff

```

`\_str_convert_encode_utf32:` Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `\_str_output_byte:n` instructions are reversed.

```

\_str_convert_encode_utf32be:
  \_str_convert_encode_utf32le:
\_str_encode_utf_xxxii_be:n      15617 \cs_new_protected:cpn { __str_convert_encode_utf32: }
  \_str_encode_utf_xxxii_be_aux:nn 15618 {
\_str_encode_utf_xxxii_le:n      15619   \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n
  \_str_encode_utf_xxxii_le_aux:nn 15620   \tl_gput_left:Ne \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
15621   }
15622 \cs_new_protected:cpn { __str_convert_encode_utf32be: }
15623   { \_str_convert_gmap_internal:N \_str_encode_utf_xxxii_be:n }
15624 \cs_new_protected:cpn { __str_convert_encode_utf32le: }

```

```

15625 { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
15626 \cs_new:Npn \__str_encode_utf_xxxii_be:n #1
15627 {
15628   \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
15629   { \int_div_truncate:nn {#1} { "100 } } {#1}
15630 }
15631 \cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
15632 {
15633   ^^00
15634   \__str_output_byte_pair_be:n {#1}
15635   \__str_output_byte:n { #2 - #1 * "100 }
15636 }
15637 \cs_new:Npn \__str_encode_utf_xxxii_le:n #1
15638 {
15639   \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
15640   { \int_div_truncate:nn {#1} { "100 } } {#1}
15641 }
15642 \cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
15643 {
15644   \__str_output_byte:n { #2 - #1 * "100 }
15645   \__str_output_byte_pair_le:n {#1}
15646   ^^00
15647 }

```

(End of definition for __str_convert_encode_utf32: and others.)

`__str_overflow` There can be no error when encoding in UTF-32. When decoding, the string may not
`__str_end` have length $4n$, or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

15648 \flag_clear_new:N \l__str_overflow_flag
15649 \flag_clear_new:N \l__str_end_flag
15650 \msg_new:nnnn { str } { utf32-decode }
15651 {
15652   Invalid-UTF-32~string:
15653   \exp_last_unbraced:Nf \use_none:n
15654   {
15655     \__str_if_flag_times:NT \l__str_overflow_flag { ,~code~point~too~large }
15656     \__str_if_flag_times:NT \l__str_end_flag      { ,~truncated~string }
15657   }
15658   .
15659 }
15660 {
15661   In~the~UTF-32~encoding,~every~Unicode~character~
15662   (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
15663   \flag_if_raised:NT \l__str_overflow_flag
15664   {
15665     \\\
15666     LaTeX~came~across~a~code~point~larger~than~1114111,~
15667     the~maximum~code~point~defined~by~Unicode.~
15668     Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
15669   }
15670   \flag_if_raised:NT \l__str_end_flag
15671   {

```

```

15672         \\\
15673         The~length~of~the~string~is~not~a~multiple~of~4.~
15674         Perhaps~the~string~was~truncated?
15675     }
15676 }

```

(End of definition for `__str_overflow` and `__str_end`.)

`__str_convert_decode_utf32`: The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `s__str_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an e-expanding assignment to `g__str_result_t1`.

The `_loop` auxiliary first checks for the end-of-string marker `s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the `(4 bytes)` `s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `s__str_stop`. Break the map.

```

15677 \cs_new_protected:cpn { __str_convert_decode_utf32be: }
15678 { \__str_decode_utf_xxxii:Nw 1 \g__str_result_t1 \s__str_stop }
15679 \cs_new_protected:cpn { __str_convert_decode_utf32le: }
15680 { \__str_decode_utf_xxxii:Nw 2 \g__str_result_t1 \s__str_stop }
15681 \cs_new_protected:cpn { __str_convert_decode_utf32: }
15682 {
15683     \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_t1
15684     \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15685 }
15686 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
15687 {
15688     \str_if_eq:nnTF { #1#2#3#4 } { ^ff ^fe ^00 ^00 }
15689     { \__str_decode_utf_xxxii:Nw 2 }
15690     {
15691         \str_if_eq:nnTF { #1#2#3#4 } { ^00 ^00 ^fe ^ff }
15692         { \__str_decode_utf_xxxii:Nw 1 }
15693         { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
15694     }
15695 }
15696 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
15697 {
15698     \flag_clear:N \l__str_overflow_flag
15699     \flag_clear:N \l__str_end_flag
15700     \flag_clear:N \l__str_error_flag
15701     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
15702     \__kernel_t1_gset:Nx \g__str_result_t1
15703     {
15704         \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
15705         #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15706         \prg_break_point:

```

```

15707     }
15708     \__str_if_flag_error:Nne \l__str_error_flag { utf32-decode } { }
15709   }
15710   \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
15711   {
15712     \if_meaning:w \s__str_stop #4
15713     \exp_after:wN \__str_decode_utf_xxxii_end:w
15714     \fi:
15715     #1#2#3#4 \s__str
15716     \if_int_compare:w \__str_tmp:w #1#4 > \c_zero_int
15717       \flag_raise:N \l__str_overflow_flag
15718       \flag_raise:N \l__str_error_flag
15719       \int_use:N \c__str_replacement_char_int
15720     \else:
15721       \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
15722         \flag_raise:N \l__str_overflow_flag
15723         \flag_raise:N \l__str_error_flag
15724         \int_use:N \c__str_replacement_char_int
15725       \else:
15726         \int_eval:n
15727         { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
15728       \fi:
15729     \fi:
15730     \s__str
15731     \__str_decode_utf_xxxii_loop:NNNN
15732   }
15733   \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
15734   {
15735     \tl_if_empty:nF {#1}
15736     {
15737       \flag_raise:N \l__str_end_flag
15738       \flag_raise:N \l__str_error_flag
15739       #1 \s__str
15740       \int_use:N \c__str_replacement_char_int \s__str
15741     }
15742     \prg_break:
15743   }

```

(End of definition for __str_convert_decode_utf32: and others.)

Restore the original catcodes of bytes 0, 254 and 255.

```

15744 \group_end:

```

60.7 PDF names and strings by expansion

```

\str_convert_pdfname:n
\__str_convert_pdfname:n
  \_str_convert_pdfname_bytes:n
  \_str_convert_pdfname_bytes_aux:n
\__str_convert_pdfname_bytes_aux:nnn

```

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

15745 \cs_new:Npn \str_convert_pdfname:n #1
15746 {
15747   \exp_args:Ne \tl_to_str:n

```



```

15748     { \str_map_function:nN {#1} \__str_convert_pdfname:n }
15749   }
15750 \sys_if_engine_opentype:TF
15751 {
15752   \cs_new:Npn \__str_convert_pdfname:n #1
15753   {
15754     \int_compare:nNnTF { '#1 } > { "7F }
15755     { \__str_convert_pdfname_bytes:n {#1} }
15756     { \__str_escape_name_char:n {#1} }
15757   }
15758   \cs_new:Npn \__str_convert_pdfname_bytes:n #1
15759   {
15760     \exp_args:Ne \__str_convert_pdfname_bytes_aux:n
15761     { \__kernel_codepoint_to_bytes:n {'#1} }
15762   }
15763   \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
15764   { \__str_convert_pdfname_bytes_aux:nnnn #1 }
15765   \cs_new:Npe \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
15766   {
15767     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
15768     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
15769     \exp_not:N \tl_if_blank:nF {#3}
15770     {
15771       \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
15772       \exp_not:N \tl_if_blank:nF {#4}
15773       {
15774         \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
15775       }
15776     }
15777   }
15778 }
15779 { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }

```

(End of definition for \str_convert_pdfname:n and others. This function is documented on page 148.)

15780 </code>

60.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```

15781 <iso88591>
15782 \__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
15783 {
15784 }
15785 {
15786 }
15787 </iso88591>
15788 <iso88592>
15789 \__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
15790 {
15791   { A1 } { 0104 }
15792   { A2 } { 02D8 }

```

15793	{ A3 }	{ 0141 }
15794	{ A5 }	{ 013D }
15795	{ A6 }	{ 015A }
15796	{ A9 }	{ 0160 }
15797	{ AA }	{ 015E }
15798	{ AB }	{ 0164 }
15799	{ AC }	{ 0179 }
15800	{ AE }	{ 017D }
15801	{ AF }	{ 017B }
15802	{ B1 }	{ 0105 }
15803	{ B2 }	{ 02DB }
15804	{ B3 }	{ 0142 }
15805	{ B5 }	{ 013E }
15806	{ B6 }	{ 015B }
15807	{ B7 }	{ 02C7 }
15808	{ B9 }	{ 0161 }
15809	{ BA }	{ 015F }
15810	{ BB }	{ 0165 }
15811	{ BC }	{ 017A }
15812	{ BD }	{ 02DD }
15813	{ BE }	{ 017E }
15814	{ BF }	{ 017C }
15815	{ C0 }	{ 0154 }
15816	{ C3 }	{ 0102 }
15817	{ C5 }	{ 0139 }
15818	{ C6 }	{ 0106 }
15819	{ C8 }	{ 010C }
15820	{ CA }	{ 0118 }
15821	{ CC }	{ 011A }
15822	{ CF }	{ 010E }
15823	{ D0 }	{ 0110 }
15824	{ D1 }	{ 0143 }
15825	{ D2 }	{ 0147 }
15826	{ D5 }	{ 0150 }
15827	{ D8 }	{ 0158 }
15828	{ D9 }	{ 016E }
15829	{ DB }	{ 0170 }
15830	{ DE }	{ 0162 }
15831	{ E0 }	{ 0155 }
15832	{ E3 }	{ 0103 }
15833	{ E5 }	{ 013A }
15834	{ E6 }	{ 0107 }
15835	{ E8 }	{ 010D }
15836	{ EA }	{ 0119 }
15837	{ EC }	{ 011B }
15838	{ EF }	{ 010F }
15839	{ F0 }	{ 0111 }
15840	{ F1 }	{ 0144 }
15841	{ F2 }	{ 0148 }
15842	{ F5 }	{ 0151 }
15843	{ F8 }	{ 0159 }
15844	{ F9 }	{ 016F }
15845	{ FB }	{ 0171 }
15846	{ FE }	{ 0163 }

```

15847     { FF } { 02D9 }
15848   }
15849   {
15850   }
15851 </iso88592>
15852 (*iso88593)
15853 \_str_declare\_eight\_bit\_encoding:nnnn { iso88593 } { 384 }
15854 {
15855   { A1 } { 0126 }
15856   { A2 } { 02D8 }
15857   { A6 } { 0124 }
15858   { A9 } { 0130 }
15859   { AA } { 015E }
15860   { AB } { 011E }
15861   { AC } { 0134 }
15862   { AF } { 017B }
15863   { B1 } { 0127 }
15864   { B6 } { 0125 }
15865   { B9 } { 0131 }
15866   { BA } { 015F }
15867   { BB } { 011F }
15868   { BC } { 0135 }
15869   { BF } { 017C }
15870   { C5 } { 010A }
15871   { C6 } { 0108 }
15872   { D5 } { 0120 }
15873   { D8 } { 011C }
15874   { DD } { 016C }
15875   { DE } { 015C }
15876   { E5 } { 010B }
15877   { E6 } { 0109 }
15878   { F5 } { 0121 }
15879   { F8 } { 011D }
15880   { FD } { 016D }
15881   { FE } { 015D }
15882   { FF } { 02D9 }
15883 }
15884 {
15885   { A5 }
15886   { AE }
15887   { BE }
15888   { C3 }
15889   { D0 }
15890   { E3 }
15891   { F0 }
15892 }
15893 </iso88593>
15894 (*iso88594)
15895 \_str_declare\_eight\_bit\_encoding:nnnn { iso88594 } { 383 }
15896 {
15897   { A1 } { 0104 }
15898   { A2 } { 0138 }
15899   { A3 } { 0156 }

```

```

15900      { A5 } { 0128 }
15901      { A6 } { 013B }
15902      { A9 } { 0160 }
15903      { AA } { 0112 }
15904      { AB } { 0122 }
15905      { AC } { 0166 }
15906      { AE } { 017D }
15907      { B1 } { 0105 }
15908      { B2 } { 02DB }
15909      { B3 } { 0157 }
15910      { B5 } { 0129 }
15911      { B6 } { 013C }
15912      { B7 } { 02C7 }
15913      { B9 } { 0161 }
15914      { BA } { 0113 }
15915      { BB } { 0123 }
15916      { BC } { 0167 }
15917      { BD } { 014A }
15918      { BE } { 017E }
15919      { BF } { 014B }
15920      { C0 } { 0100 }
15921      { C7 } { 012E }
15922      { C8 } { 010C }
15923      { CA } { 0118 }
15924      { CC } { 0116 }
15925      { CF } { 012A }
15926      { D0 } { 0110 }
15927      { D1 } { 0145 }
15928      { D2 } { 014C }
15929      { D3 } { 0136 }
15930      { D9 } { 0172 }
15931      { DD } { 0168 }
15932      { DE } { 016A }
15933      { E0 } { 0101 }
15934      { E7 } { 012F }
15935      { E8 } { 010D }
15936      { EA } { 0119 }
15937      { EC } { 0117 }
15938      { EF } { 012B }
15939      { F0 } { 0111 }
15940      { F1 } { 0146 }
15941      { F2 } { 014D }
15942      { F3 } { 0137 }
15943      { F9 } { 0173 }
15944      { FD } { 0169 }
15945      { FE } { 016B }
15946      { FF } { 02D9 }
15947      }
15948      {
15949      }
15950      </iso88594>

15951      <iso88595>
15952      \__str_declare_eight_bit_encoding:nnnn { iso88595 } { 374 }
15953      {

```

15954	{ A1 }	{ 0401 }
15955	{ A2 }	{ 0402 }
15956	{ A3 }	{ 0403 }
15957	{ A4 }	{ 0404 }
15958	{ A5 }	{ 0405 }
15959	{ A6 }	{ 0406 }
15960	{ A7 }	{ 0407 }
15961	{ A8 }	{ 0408 }
15962	{ A9 }	{ 0409 }
15963	{ AA }	{ 040A }
15964	{ AB }	{ 040B }
15965	{ AC }	{ 040C }
15966	{ AE }	{ 040E }
15967	{ AF }	{ 040F }
15968	{ B0 }	{ 0410 }
15969	{ B1 }	{ 0411 }
15970	{ B2 }	{ 0412 }
15971	{ B3 }	{ 0413 }
15972	{ B4 }	{ 0414 }
15973	{ B5 }	{ 0415 }
15974	{ B6 }	{ 0416 }
15975	{ B7 }	{ 0417 }
15976	{ B8 }	{ 0418 }
15977	{ B9 }	{ 0419 }
15978	{ BA }	{ 041A }
15979	{ BB }	{ 041B }
15980	{ BC }	{ 041C }
15981	{ BD }	{ 041D }
15982	{ BE }	{ 041E }
15983	{ BF }	{ 041F }
15984	{ C0 }	{ 0420 }
15985	{ C1 }	{ 0421 }
15986	{ C2 }	{ 0422 }
15987	{ C3 }	{ 0423 }
15988	{ C4 }	{ 0424 }
15989	{ C5 }	{ 0425 }
15990	{ C6 }	{ 0426 }
15991	{ C7 }	{ 0427 }
15992	{ C8 }	{ 0428 }
15993	{ C9 }	{ 0429 }
15994	{ CA }	{ 042A }
15995	{ CB }	{ 042B }
15996	{ CC }	{ 042C }
15997	{ CD }	{ 042D }
15998	{ CE }	{ 042E }
15999	{ CF }	{ 042F }
16000	{ D0 }	{ 0430 }
16001	{ D1 }	{ 0431 }
16002	{ D2 }	{ 0432 }
16003	{ D3 }	{ 0433 }
16004	{ D4 }	{ 0434 }
16005	{ D5 }	{ 0435 }
16006	{ D6 }	{ 0436 }
16007	{ D7 }	{ 0437 }

```

16008     { D8 } { 0438 }
16009     { D9 } { 0439 }
16010     { DA } { 043A }
16011     { DB } { 043B }
16012     { DC } { 043C }
16013     { DD } { 043D }
16014     { DE } { 043E }
16015     { DF } { 043F }
16016     { E0 } { 0440 }
16017     { E1 } { 0441 }
16018     { E2 } { 0442 }
16019     { E3 } { 0443 }
16020     { E4 } { 0444 }
16021     { E5 } { 0445 }
16022     { E6 } { 0446 }
16023     { E7 } { 0447 }
16024     { E8 } { 0448 }
16025     { E9 } { 0449 }
16026     { EA } { 044A }
16027     { EB } { 044B }
16028     { EC } { 044C }
16029     { ED } { 044D }
16030     { EE } { 044E }
16031     { EF } { 044F }
16032     { F0 } { 2116 }
16033     { F1 } { 0451 }
16034     { F2 } { 0452 }
16035     { F3 } { 0453 }
16036     { F4 } { 0454 }
16037     { F5 } { 0455 }
16038     { F6 } { 0456 }
16039     { F7 } { 0457 }
16040     { F8 } { 0458 }
16041     { F9 } { 0459 }
16042     { FA } { 045A }
16043     { FB } { 045B }
16044     { FC } { 045C }
16045     { FD } { 00A7 }
16046     { FE } { 045E }
16047     { FF } { 045F }
16048     }
16049     {
16050     }
16051     </iso88595>
16052     <*iso88596>
16053     \_str_declare\_eight\_bit\_encoding:nnnn { iso88596 } { 344 }
16054     {
16055         { AC } { 060C }
16056         { BB } { 061B }
16057         { BF } { 061F }
16058         { C1 } { 0621 }
16059         { C2 } { 0622 }
16060         { C3 } { 0623 }
16061         { C4 } { 0624 }

```

```

16062      { C5 } { 0625 }
16063      { C6 } { 0626 }
16064      { C7 } { 0627 }
16065      { C8 } { 0628 }
16066      { C9 } { 0629 }
16067      { CA } { 062A }
16068      { CB } { 062B }
16069      { CC } { 062C }
16070      { CD } { 062D }
16071      { CE } { 062E }
16072      { CF } { 062F }
16073      { D0 } { 0630 }
16074      { D1 } { 0631 }
16075      { D2 } { 0632 }
16076      { D3 } { 0633 }
16077      { D4 } { 0634 }
16078      { D5 } { 0635 }
16079      { D6 } { 0636 }
16080      { D7 } { 0637 }
16081      { D8 } { 0638 }
16082      { D9 } { 0639 }
16083      { DA } { 063A }
16084      { E0 } { 0640 }
16085      { E1 } { 0641 }
16086      { E2 } { 0642 }
16087      { E3 } { 0643 }
16088      { E4 } { 0644 }
16089      { E5 } { 0645 }
16090      { E6 } { 0646 }
16091      { E7 } { 0647 }
16092      { E8 } { 0648 }
16093      { E9 } { 0649 }
16094      { EA } { 064A }
16095      { EB } { 064B }
16096      { EC } { 064C }
16097      { ED } { 064D }
16098      { EE } { 064E }
16099      { EF } { 064F }
16100      { F0 } { 0650 }
16101      { F1 } { 0651 }
16102      { F2 } { 0652 }
16103      }
16104      {
16105          { A1 }
16106          { A2 }
16107          { A3 }
16108          { A5 }
16109          { A6 }
16110          { A7 }
16111          { A8 }
16112          { A9 }
16113          { AA }
16114          { AB }
16115          { AE }

```

```

16116     { AF }
16117     { B0 }
16118     { B1 }
16119     { B2 }
16120     { B3 }
16121     { B4 }
16122     { B5 }
16123     { B6 }
16124     { B7 }
16125     { B8 }
16126     { B9 }
16127     { BA }
16128     { BC }
16129     { BD }
16130     { BE }
16131     { CO }
16132     { DB }
16133     { DC }
16134     { DD }
16135     { DE }
16136     { DF }
16137 }
16138 </iso88596>
16139 <*iso88597>
16140 \__str_declare_eight_bit_encoding:nmmn { iso88597 } { 498 }
16141 {
16142     { A1 } { 2018 }
16143     { A2 } { 2019 }
16144     { A4 } { 20AC }
16145     { A5 } { 20AF }
16146     { AA } { 037A }
16147     { AF } { 2015 }
16148     { B4 } { 0384 }
16149     { B5 } { 0385 }
16150     { B6 } { 0386 }
16151     { B8 } { 0388 }
16152     { B9 } { 0389 }
16153     { BA } { 038A }
16154     { BC } { 038C }
16155     { BE } { 038E }
16156     { BF } { 038F }
16157     { C0 } { 0390 }
16158     { C1 } { 0391 }
16159     { C2 } { 0392 }
16160     { C3 } { 0393 }
16161     { C4 } { 0394 }
16162     { C5 } { 0395 }
16163     { C6 } { 0396 }
16164     { C7 } { 0397 }
16165     { C8 } { 0398 }
16166     { C9 } { 0399 }
16167     { CA } { 039A }
16168     { CB } { 039B }
16169     { CC } { 039C }

```



```

16170      { CD } { 039D }
16171      { CE } { 039E }
16172      { CF } { 039F }
16173      { D0 } { 03A0 }
16174      { D1 } { 03A1 }
16175      { D3 } { 03A3 }
16176      { D4 } { 03A4 }
16177      { D5 } { 03A5 }
16178      { D6 } { 03A6 }
16179      { D7 } { 03A7 }
16180      { D8 } { 03A8 }
16181      { D9 } { 03A9 }
16182      { DA } { 03AA }
16183      { DB } { 03AB }
16184      { DC } { 03AC }
16185      { DD } { 03AD }
16186      { DE } { 03AE }
16187      { DF } { 03AF }
16188      { E0 } { 03B0 }
16189      { E1 } { 03B1 }
16190      { E2 } { 03B2 }
16191      { E3 } { 03B3 }
16192      { E4 } { 03B4 }
16193      { E5 } { 03B5 }
16194      { E6 } { 03B6 }
16195      { E7 } { 03B7 }
16196      { E8 } { 03B8 }
16197      { E9 } { 03B9 }
16198      { EA } { 03BA }
16199      { EB } { 03BB }
16200      { EC } { 03BC }
16201      { ED } { 03BD }
16202      { EE } { 03BE }
16203      { EF } { 03BF }
16204      { F0 } { 03C0 }
16205      { F1 } { 03C1 }
16206      { F2 } { 03C2 }
16207      { F3 } { 03C3 }
16208      { F4 } { 03C4 }
16209      { F5 } { 03C5 }
16210      { F6 } { 03C6 }
16211      { F7 } { 03C7 }
16212      { F8 } { 03C8 }
16213      { F9 } { 03C9 }
16214      { FA } { 03CA }
16215      { FB } { 03CB }
16216      { FC } { 03CC }
16217      { FD } { 03CD }
16218      { FE } { 03CE }
16219      }
16220      {
16221          { AE }
16222          { D2 }
16223      }

```

```

16224 </iso88597>
16225 <*iso88598>
16226 \_str_declare_eight_bit_encoding:nnnn { iso88598 } { 308 }
16227 {
16228     { AA } { 00D7 }
16229     { BA } { 00F7 }
16230     { DF } { 2017 }
16231     { E0 } { 05D0 }
16232     { E1 } { 05D1 }
16233     { E2 } { 05D2 }
16234     { E3 } { 05D3 }
16235     { E4 } { 05D4 }
16236     { E5 } { 05D5 }
16237     { E6 } { 05D6 }
16238     { E7 } { 05D7 }
16239     { E8 } { 05D8 }
16240     { E9 } { 05D9 }
16241     { EA } { 05DA }
16242     { EB } { 05DB }
16243     { EC } { 05DC }
16244     { ED } { 05DD }
16245     { EE } { 05DE }
16246     { EF } { 05DF }
16247     { F0 } { 05E0 }
16248     { F1 } { 05E1 }
16249     { F2 } { 05E2 }
16250     { F3 } { 05E3 }
16251     { F4 } { 05E4 }
16252     { F5 } { 05E5 }
16253     { F6 } { 05E6 }
16254     { F7 } { 05E7 }
16255     { F8 } { 05E8 }
16256     { F9 } { 05E9 }
16257     { FA } { 05EA }
16258     { FD } { 200E }
16259     { FE } { 200F }
16260 }
16261 {
16262     { A1 }
16263     { BF }
16264     { C0 }
16265     { C1 }
16266     { C2 }
16267     { C3 }
16268     { C4 }
16269     { C5 }
16270     { C6 }
16271     { C7 }
16272     { C8 }
16273     { C9 }
16274     { CA }
16275     { CB }
16276     { CC }
16277     { CD }

```

```

16278     { CE }
16279     { CF }
16280     { D0 }
16281     { D1 }
16282     { D2 }
16283     { D3 }
16284     { D4 }
16285     { D5 }
16286     { D6 }
16287     { D7 }
16288     { D8 }
16289     { D9 }
16290     { DA }
16291     { DB }
16292     { DC }
16293     { DD }
16294     { DE }
16295     { FB }
16296     { FC }
16297 }
16298 </iso88598>
16299 <*iso88599>
16300 \_str_declare\_eight\_bit\_encoding:nnnn { iso88599 } { 352 }
16301 {
16302     { D0 } { 011E }
16303     { DD } { 0130 }
16304     { DE } { 015E }
16305     { FO } { 011F }
16306     { FD } { 0131 }
16307     { FE } { 015F }
16308 }
16309 {
16310 }
16311 </iso88599>
16312 <*iso885910>
16313 \_str_declare\_eight\_bit\_encoding:nnnn { iso885910 } { 383 }
16314 {
16315     { A1 } { 0104 }
16316     { A2 } { 0112 }
16317     { A3 } { 0122 }
16318     { A4 } { 012A }
16319     { A5 } { 0128 }
16320     { A6 } { 0136 }
16321     { A8 } { 013B }
16322     { A9 } { 0110 }
16323     { AA } { 0160 }
16324     { AB } { 0166 }
16325     { AC } { 017D }
16326     { AE } { 016A }
16327     { AF } { 014A }
16328     { B1 } { 0105 }
16329     { B2 } { 0113 }
16330     { B3 } { 0123 }

```

```

16331     { B4 } { 012B }
16332     { B5 } { 0129 }
16333     { B6 } { 0137 }
16334     { B8 } { 013C }
16335     { B9 } { 0111 }
16336     { BA } { 0161 }
16337     { BB } { 0167 }
16338     { BC } { 017E }
16339     { BD } { 2015 }
16340     { BE } { 016B }
16341     { BF } { 014B }
16342     { C0 } { 0100 }
16343     { C7 } { 012E }
16344     { C8 } { 010C }
16345     { CA } { 0118 }
16346     { CC } { 0116 }
16347     { D1 } { 0145 }
16348     { D2 } { 014C }
16349     { D7 } { 0168 }
16350     { D9 } { 0172 }
16351     { E0 } { 0101 }
16352     { E7 } { 012F }
16353     { E8 } { 010D }
16354     { EA } { 0119 }
16355     { EC } { 0117 }
16356     { F1 } { 0146 }
16357     { F2 } { 014D }
16358     { F7 } { 0169 }
16359     { F9 } { 0173 }
16360     { FF } { 0138 }
16361 }
16362 {
16363 }
16364 </iso885910>
16365 <*iso885911>
16366 \__str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
16367 {
16368     { A1 } { 0E01 }
16369     { A2 } { 0E02 }
16370     { A3 } { 0E03 }
16371     { A4 } { 0E04 }
16372     { A5 } { 0E05 }
16373     { A6 } { 0E06 }
16374     { A7 } { 0E07 }
16375     { A8 } { 0E08 }
16376     { A9 } { 0E09 }
16377     { AA } { 0E0A }
16378     { AB } { 0E0B }
16379     { AC } { 0E0C }
16380     { AD } { 0E0D }
16381     { AE } { 0E0E }
16382     { AF } { 0E0F }
16383     { B0 } { 0E10 }
16384     { B1 } { 0E11 }

```

16385	{ B2 }	{ 0E12 }
16386	{ B3 }	{ 0E13 }
16387	{ B4 }	{ 0E14 }
16388	{ B5 }	{ 0E15 }
16389	{ B6 }	{ 0E16 }
16390	{ B7 }	{ 0E17 }
16391	{ B8 }	{ 0E18 }
16392	{ B9 }	{ 0E19 }
16393	{ BA }	{ 0E1A }
16394	{ BB }	{ 0E1B }
16395	{ BC }	{ 0E1C }
16396	{ BD }	{ 0E1D }
16397	{ BE }	{ 0E1E }
16398	{ BF }	{ 0E1F }
16399	{ C0 }	{ 0E20 }
16400	{ C1 }	{ 0E21 }
16401	{ C2 }	{ 0E22 }
16402	{ C3 }	{ 0E23 }
16403	{ C4 }	{ 0E24 }
16404	{ C5 }	{ 0E25 }
16405	{ C6 }	{ 0E26 }
16406	{ C7 }	{ 0E27 }
16407	{ C8 }	{ 0E28 }
16408	{ C9 }	{ 0E29 }
16409	{ CA }	{ 0E2A }
16410	{ CB }	{ 0E2B }
16411	{ CC }	{ 0E2C }
16412	{ CD }	{ 0E2D }
16413	{ CE }	{ 0E2E }
16414	{ CF }	{ 0E2F }
16415	{ D0 }	{ 0E30 }
16416	{ D1 }	{ 0E31 }
16417	{ D2 }	{ 0E32 }
16418	{ D3 }	{ 0E33 }
16419	{ D4 }	{ 0E34 }
16420	{ D5 }	{ 0E35 }
16421	{ D6 }	{ 0E36 }
16422	{ D7 }	{ 0E37 }
16423	{ D8 }	{ 0E38 }
16424	{ D9 }	{ 0E39 }
16425	{ DA }	{ 0E3A }
16426	{ DF }	{ 0E3F }
16427	{ E0 }	{ 0E40 }
16428	{ E1 }	{ 0E41 }
16429	{ E2 }	{ 0E42 }
16430	{ E3 }	{ 0E43 }
16431	{ E4 }	{ 0E44 }
16432	{ E5 }	{ 0E45 }
16433	{ E6 }	{ 0E46 }
16434	{ E7 }	{ 0E47 }
16435	{ E8 }	{ 0E48 }
16436	{ E9 }	{ 0E49 }
16437	{ EA }	{ 0E4A }
16438	{ EB }	{ 0E4B }

```

16439     { EC } { 0E4C }
16440     { ED } { 0E4D }
16441     { EE } { 0E4E }
16442     { EF } { 0E4F }
16443     { F0 } { 0E50 }
16444     { F1 } { 0E51 }
16445     { F2 } { 0E52 }
16446     { F3 } { 0E53 }
16447     { F4 } { 0E54 }
16448     { F5 } { 0E55 }
16449     { F6 } { 0E56 }
16450     { F7 } { 0E57 }
16451     { F8 } { 0E58 }
16452     { F9 } { 0E59 }
16453     { FA } { 0E5A }
16454     { FB } { 0E5B }
16455 }
16456 {
16457     { DB }
16458     { DC }
16459     { DD }
16460     { DE }
16461 }
16462 </iso885911>
16463 <*iso885913>
16464 \_str_declare\_eight\_bit\_encoding:nnnn { iso885913 } { 399 }
16465 {
16466     { A1 } { 201D }
16467     { A5 } { 201E }
16468     { A8 } { 00D8 }
16469     { AA } { 0156 }
16470     { AF } { 00C6 }
16471     { B4 } { 201C }
16472     { B8 } { 00F8 }
16473     { BA } { 0157 }
16474     { BF } { 00E6 }
16475     { C0 } { 0104 }
16476     { C1 } { 012E }
16477     { C2 } { 0100 }
16478     { C3 } { 0106 }
16479     { C6 } { 0118 }
16480     { C7 } { 0112 }
16481     { C8 } { 010C }
16482     { CA } { 0179 }
16483     { CB } { 0116 }
16484     { CC } { 0122 }
16485     { CD } { 0136 }
16486     { CE } { 012A }
16487     { CF } { 013B }
16488     { D0 } { 0160 }
16489     { D1 } { 0143 }
16490     { D2 } { 0145 }
16491     { D4 } { 014C }
16492     { D8 } { 0172 }

```

```

16493     { D9 } { 0141 }
16494     { DA } { 015A }
16495     { DB } { 016A }
16496     { DD } { 017B }
16497     { DE } { 017D }
16498     { EO } { 0105 }
16499     { E1 } { 012F }
16500     { E2 } { 0101 }
16501     { E3 } { 0107 }
16502     { E6 } { 0119 }
16503     { E7 } { 0113 }
16504     { E8 } { 010D }
16505     { EA } { 017A }
16506     { EB } { 0117 }
16507     { EC } { 0123 }
16508     { ED } { 0137 }
16509     { EE } { 012B }
16510     { EF } { 013C }
16511     { FO } { 0161 }
16512     { F1 } { 0144 }
16513     { F2 } { 0146 }
16514     { F4 } { 014D }
16515     { F8 } { 0173 }
16516     { F9 } { 0142 }
16517     { FA } { 015B }
16518     { FB } { 016B }
16519     { FD } { 017C }
16520     { FE } { 017E }
16521     { FF } { 2019 }
16522 }
16523 {
16524 }
16525 </iso885913>
16526 <*iso885914>
16527 \_str_declare\_eight\_bit\_encoding:nnnn { iso885914 } { 529 }
16528 {
16529     { A1 } { 1E02 }
16530     { A2 } { 1E03 }
16531     { A4 } { 010A }
16532     { A5 } { 010B }
16533     { A6 } { 1E0A }
16534     { A8 } { 1E80 }
16535     { AA } { 1E82 }
16536     { AB } { 1E0B }
16537     { AC } { 1EF2 }
16538     { AF } { 0178 }
16539     { B0 } { 1E1E }
16540     { B1 } { 1E1F }
16541     { B2 } { 0120 }
16542     { B3 } { 0121 }
16543     { B4 } { 1E40 }
16544     { B5 } { 1E41 }
16545     { B7 } { 1E56 }
16546     { B8 } { 1E81 }

```

```

16547     { B9 } { 1E57 }
16548     { BA } { 1E83 }
16549     { BB } { 1E60 }
16550     { BC } { 1EF3 }
16551     { BD } { 1E84 }
16552     { BE } { 1E85 }
16553     { BF } { 1E61 }
16554     { D0 } { 0174 }
16555     { D7 } { 1E6A }
16556     { DE } { 0176 }
16557     { F0 } { 0175 }
16558     { F7 } { 1E6B }
16559     { FE } { 0177 }
16560   }
16561   {
16562   }
16563 </iso885914>
16564 (*iso885915)
16565 \_str_declare\_eight\_bit\_encoding:nnnn { iso885915 } { 383 }
16566   {
16567     { A4 } { 20AC }
16568     { A6 } { 0160 }
16569     { A8 } { 0161 }
16570     { B4 } { 017D }
16571     { B8 } { 017E }
16572     { BC } { 0152 }
16573     { BD } { 0153 }
16574     { BE } { 0178 }
16575   }
16576   {
16577   }
16578 </iso885915>
16579 (*iso885916)
16580 \_str_declare\_eight\_bit\_encoding:nnnn { iso885916 } { 558 }
16581   {
16582     { A1 } { 0104 }
16583     { A2 } { 0105 }
16584     { A3 } { 0141 }
16585     { A4 } { 20AC }
16586     { A5 } { 201E }
16587     { A6 } { 0160 }
16588     { A8 } { 0161 }
16589     { AA } { 0218 }
16590     { AC } { 0179 }
16591     { AE } { 017A }
16592     { AF } { 017B }
16593     { B2 } { 010C }
16594     { B3 } { 0142 }
16595     { B4 } { 017D }
16596     { B5 } { 201D }
16597     { B8 } { 017E }
16598     { B9 } { 010D }
16599     { BA } { 0219 }

```



```

16600      { BC } { 0152 }
16601      { BD } { 0153 }
16602      { BE } { 0178 }
16603      { BF } { 017C }
16604      { C3 } { 0102 }
16605      { C5 } { 0106 }
16606      { D0 } { 0110 }
16607      { D1 } { 0143 }
16608      { D5 } { 0150 }
16609      { D7 } { 015A }
16610      { D8 } { 0170 }
16611      { DD } { 0118 }
16612      { DE } { 021A }
16613      { E3 } { 0103 }
16614      { E5 } { 0107 }
16615      { F0 } { 0111 }
16616      { F1 } { 0144 }
16617      { F5 } { 0151 }
16618      { F7 } { 015B }
16619      { F8 } { 0171 }
16620      { FD } { 0119 }
16621      { FE } { 021B }
16622      }
16623      {
16624      }
16625      </iso885916>

```

Chapter 61

l3str-format implementation

16626 `\*code)`

16627 `\@@=str)`

61.1 Helpers

`\use:nf` A simple variant.

`\use:fnnf` 16628 `\cs_generate_variant:Nn \use:nn { nf }`
16629 `\cs_generate_variant:Nn \use:nnn { fnf }`

(End of definition for \use:nf and \use:fnnf. These functions are documented on page ??.)

`\tl_to_str:f` A simple variant.

16630 `\cs_generate_variant:Nn \tl_to_str:n { f }`

(End of definition for \tl_to_str:f. This function is documented on page ??.)

`\__str_format_if_digit:NTF` Here we expect #1 to be a character with category other, or `\s__str_stop`.

16631 `\prg_new_conditional:Npnn \__str_format_if_digit:N #1 { TF }`
16632 `{`
16633 `\if_int_compare:w 9 < 1 #1 \exp_stop_f:`
16634 `\prg_return_true: \else: \prg_return_false: \fi:`
16635 `}`

(End of definition for __str_format_if_digit:NTF.)

`\__str_format_put:nw` Put #1 after an `\s__str_stop` delimiter.

`\__str_format_put:ow` 16636 `\cs_new:Npn \__str_format_put:nw #1 #2 \s__str_stop { #2 \s__str_stop #1 }`
`\__str_format_put:fw` 16637 `\cs_generate_variant:Nn \__str_format_put:nw { o , f }`

(End of definition for __str_format_put:nw.)

`\__str_format_if_in:nNTF` A copy of `\__str_if_contains_char:nNTF` to avoid relying on this weird internal string
`\__str_format_if_in_aux:NN` function.

16638 `\prg_new_conditional:Npnn \__str_format_if_in:nN #1#2 { TF }`
16639 `{`
16640 `\__str_format_if_in_aux:NN #2 #1`
16641 `{ #2 \prg_return_false: \exp_after:wN \prg_break: \else: }`
16642 `\prg_break_point:`

```

16643 }
16644 \cs_new:Npn \__str_format_if_in_aux:NN #1#2
16645 {
16646   \if_charcode:w #1 #2
16647     \prg_return_true:
16648     \exp_after:wN \prg_break:
16649   \fi:
16650   \__str_format_if_in_aux:NN #1
16651 }

```

(End of definition for __str_format_if_in:nNTF and __str_format_if_in_aux:NN.)

61.2 Parsing a format specification

The goal is to parse

$\langle \textit{format specification} \rangle = [[\langle \textit{fill} \rangle]] [\langle \textit{alignment} \rangle] [\langle \textit{sign} \rangle] [\langle \textit{width} \rangle] [\langle \textit{precision} \rangle] [\langle \textit{style} \rangle]$

```

\__str_format_parse:n
\__str_format_parse_auxi:NN
\__str_format_parse_auxii:nN
  \__str_format_parse_auxiii:nN
  \__str_format_parse_auxiv:nwN
\__str_format_parse_auxv:nN
  \__str_format_parse_auxvi:nwN
  \__str_format_parse_auxvii:nN
\__str_format_parse_end:nwn
16652 \cs_new:Npn \__str_format_parse:n #1
16653 {
16654   \exp_last_unbraced:Nf \__str_format_parse_auxi:NN
16655   { \__kernel_str_to_other:n {#1} } \s__str_stop \s__str_stop {#1}
16656 }
16657 \cs_new:Npe \__str_format_parse_auxi:NN #1#2
16658 {
16659   \exp_not:N \__str_format_if_in:nNTF { < > = ^ } #2
16660   { \exp_not:N \__str_format_parse_auxiii:nN { #1 #2 } }
16661   {
16662     \exp_not:N \__str_format_parse_auxii:nN
16663     { \c_catcode_other_space_tl } #1 #2
16664   }
16665 }
16666 \cs_new:Npn \__str_format_parse_auxii:nN #1#2
16667 {
16668   \__str_format_if_in:nNTF { < > = ^ } #2
16669   { \__str_format_parse_auxiii:nN { #1 #2 } }
16670   { \__str_format_parse_auxiii:nN { #1 ? } #2 }
16671 }
16672 \cs_new:Npe \__str_format_parse_auxiii:nN #1#2
16673 {
16674   \exp_not:N \__str_format_if_in:nNTF
16675   { + - \c_catcode_other_space_tl }
16676   #2
16677   { \exp_not:N \__str_format_parse_auxiv:nwN { #1 #2 } ; }
16678   { \exp_not:N \__str_format_parse_auxiv:nwN { #1 ? } ; #2 }
16679 }
16680 \cs_new:Npn \__str_format_parse_auxiv:nwN #1#2; #3
16681 {
16682   \__str_format_if_digit:NTF #3
16683   { \__str_format_parse_auxiv:nwN {#1} #2 #3 ; }
16684   { \__str_format_parse_auxv:nN { #1 {#2} } #3 }
16685 }
16686 \cs_new:Npn \__str_format_parse_auxv:nN #1#2
16687 {

```

```

16688     \token_if_eq_charcode:NNTF . #2
16689     { \__str_format_parse_auxvi:nwN {#1} 0 ; }
16690     { \__str_format_parse_auxvii:nN { #1 { } } #2 }
16691   }
16692 \cs_new:Npn \__str_format_parse_auxvi:nwN #1#2; #3
16693 {
16694   \__str_format_if_digit:NTF #3
16695   { \__str_format_parse_auxvi:nwN {#1} #2 #3 ; }
16696   { \__str_format_parse_auxvii:nN { #1 {#2} } #3 }
16697 }
16698 \cs_new:Npn \__str_format_parse_auxvii:nN #1#2
16699 {
17000   \token_if_eq_meaning:NNTF \s__str_stop #2
17001   { \__str_format_parse_end:nwn { #1 ? } #2 }
17002   { \__str_format_parse_end:nwn { #1 #2 } }
17003 }
17004 \cs_new:Npn \__str_format_parse_end:nwn #1 #2 \s__str_stop \s__str_stop #3
17005 {
17006   \tl_if_empty:nF {#2}
17007   { \msg_expandable_error:nnn { str } { invalid-format } {#3} }
17008   #1
17009 }

```

(End of definition for `\__str_format_parse:n` and others.)

61.3 Alignment

The 4 functions in this section receive an $\langle body \rangle$, a $\langle sign \rangle$, a $\langle width \rangle$ and a $\langle fill \rangle$ character (exactly one character). For non-numeric types, the $\langle sign \rangle$ is empty and the $\langle body \rangle$ is the (other) string we want to format. For numeric types, we wish to format $\langle sign \rangle \langle body \rangle$ (both are other strings). The alignment types `<`, `>` and `^` keep $\langle sign \rangle$ and $\langle body \rangle$ together. The `=` alignment type, however, inserts the padding between the $\langle sign \rangle$ and the $\langle body \rangle$, hence the need to keep those separate.

```

\__str_format_align_<:nnnN \__str_format_align_<:nnnN {<body>} {<sign>} {<width>} {<fill>}

```

Aligning “ $\langle sign \rangle \langle body \rangle$ ” to the left entails appending #4 the correct number of times. Then convert the result to a string.

```

16710 \cs_new:cpn { \__str_format_align_<:nnnN } #1#2#3#4
16711 {
16712   \use:nf { #2 #1 }
16713   {
16714     \prg_replicate:nn
16715     { \int_max:nn { #3 - \__str_count:n { #2 #1 } } { 0 } }
16716     {#4}
16717   }
16718 }

```

(End of definition for `\__str_format_align_<:nnnN`.)

```

\__str_format_align_>:nnnN \__str_format_align_>:nnnN {<body>} {<sign>} {<width>} {<fill>}

```

Aligning an “ $\langle sign \rangle \langle body \rangle$ ” to the right entails prepending #4 the correct number of times. Then convert the result to a string.

```

16719 \cs_new:cpn { \__str_format_align_>:nnnN } #1#2#3#4

```

```

16720 {
16721   \prg_replicate:nn
16722   { \int_max:nn { #3 - \_str_count:n { #2 #1 } } { 0 } }
16723   {#4}
16724   #2 #1
16725 }

```

(End of definition for _str_format_align_>:nnnN.)

_str_format_align^:nnnN

_str_format_align^:nnnN {<body>} {<sign>} {<width>} {<fill>}

Centering “<sign> <body>” entails prepending and appending #4 the correct number of times. If the number of #4 to be added is odd, we add one more after than before.

```

16726 \cs_new:cpn { \_str_format_align^:nnnN } #1#2#3#4
16727 {
16728   \use:nfn
16729   {
16730     \prg_replicate:nn
16731     {
16732       \int_max:nn { 0 }
16733       { #3 - \_str_count:n { #2 #1 } - 1 }
16734       / 2
16735     }
16736     {#4}
16737   }
16738   { #2 #1 }
16739   {
16740     \prg_replicate:nn
16741     {
16742       \int_max:nn { 0 }
16743       { #3 - \_str_count:n { #2 #1 } }
16744       / 2
16745     }
16746     {#4}
16747   }
16748 }

```

(End of definition for _str_format_align^:nnnN.)

_str_format_align=:nnnN

_str_format_align=:nnnN {<body>} {<sign>} {<width>} {<fill>}

The special numeric alignment = means that we insert the appropriate number of copies of #4 between the <sign> and the <body>. Then convert the result to a string.

```

16749 \cs_new:cpn { \_str_format_align=:nnnN } #1#2#3#4
16750 {
16751   \use:nf {#2}
16752   {
16753     \prg_replicate:nn
16754     { \int_max:nn { #3 - \_str_count:n { #2 #1 } } { 0 } }
16755     {#4}
16756   }
16757   #1
16758 }

```

(End of definition for _str_format_align=:nnnN.)

61.4 Formatting token lists

`\tl_format:Nn` Call `\__str_format_tl:NNNnnNn` to read the parsed *<format specification>*. Then
`\tl_format:cn` convert the result to a string.

`\tl_format:nn`

```

16759 \cs_new:Npn \tl_format:Nn { \exp_args:No \tl_format:nn }
16760 \cs_generate_variant:Nn \tl_format:Nn { c }
16761 \cs_new:Npn \tl_format:nn #1#2
16762 {
16763   \tl_to_str:f
16764   {
16765     \exp_last_unbraced:Nf \__str_format_tl:NNNnnNn
16766     { \__str_format_parse:n {#2} }
16767     {#1}
16768   }
16769 }
```

(End of definition for `\tl_format:Nn` and `\tl_format:nn`. These functions are documented on page 127.)

`\__str_format_tl:NNNnnNn`

```

\__str_format_tl:NNNnnNn <fill> <alignment> <sign> {<width>} {<precision>}
<style> {<token list>}
```

First check that the *<alignment>* is not =, and set the default alignment ? to <. Place the modified information after a trailing `\s__str_stop` for later retrieval. Then check that there was no *<sign>*. The width will be useful later, store it after `\s__str_stop`. Afterwards, store the precision, and the function `\__str_range:nnn` that will be used to extract the first #5 characters of the string. There is a need to use the internal function, as otherwise leading spaces would get stripped by f-expansion. Finally, check that the *<style>* is ? or s.

```

16770 \cs_new:Npn \__str_format_tl:NNNnnNn #1#2#3#4#5#6
16771 {
16772   \token_if_eq_charcode:NNTF #2 =
16773   {
16774     \msg_expandable_error:nnnn
16775     { str } { invalid-align-format } {#2} {t1}
16776     \__str_format_put:nw { #1 < }
16777   }
16778   {
16779     \token_if_eq_charcode:NNTF #2 ?
16780     { \__str_format_put:nw { #1 < } }
16781     { \__str_format_put:nw { #1 #2 } }
16782   }
16783   \token_if_eq_charcode:NNTF #3 ?
16784   {
16785     \msg_expandable_error:nnnn
16786     { str } { invalid-sign-format } {#3} {t1}
16787   }
16788   \__str_format_put:nw { {#4} }
16789   \tl_if_empty:nTF {#5}
16790   { \__str_format_put:nw { \__str_range:nnn { {1} {-1} } } }
16791   { \__str_format_put:nw { \__str_range:nnn { {1} {#5} } } }
16792   \token_if_eq_charcode:NNTF #6 s
16793   {
16794     \token_if_eq_charcode:NNTF #6 ?
```

```

16795         {
16796             \msg_expandable_error:nnnn
16797             { str } { invalid-style-format } {#6} {tl}
16798         }
16799     }
16800     \__str_format_tl_s:NNnnNNn
16801     \s__str_stop
16802 }

```

(End of definition for __str_format_tl:NNnnNNn.)

```

\__str_format_tl_s:NNnnNNn    \__str_format_tl_s:NNnnNNn \s__str_stop <function> {<arguments>} {<width>}
                              <fill> <alignment> {<token list>}

```

The *<function>* and *<arguments>* are built in such a way that f-expanding *<function> {<other string>}* *<arguments>* yields the piece of the *<other string>* that we want to output. The *<other string>* is built from the *<token list>* by f-expanding *__kernel_str_to_other:n*.

```

16803 \cs_new:Npn \__str_format_tl_s:NNnnNNn #1#2#3#4#5#6#7
16804 {
16805     \exp_args:Nc \exp_args:Nf
16806     { __str_format_align_#6:nnnN }
16807     { \exp_args:Nf #2 { \__kernel_str_to_other:n {#7} } } #3 }
16808     { }
16809     {#4} #5
16810 }

```

(End of definition for __str_format_tl_s:NNnnNNn.)

61.5 Formatting sequences

\seq_format:Nn Each item is formatted as a token list according to the specification. First parse the
\seq_format:cn format and expand the sequence, then loop through the items. Eventually, convert to a string.

```

16811 \cs_new:Npn \seq_format:Nn #1#2
16812 {
16813     \tl_to_str:f
16814     {
16815         \__str_format_seq:ff
16816         { \exp_after:wN \use_i:nn \exp_after:wN \exp_stop_f: #1 }
16817         { \__str_format_parse:n {#2} }
16818     }
16819 }
16820 \cs_generate_variant:Nn \seq_format:Nn { c }

```

(End of definition for \seq_format:Nn. This function is documented on page 167.)

__str_format_seq:nn The first argument is the contents of a *seq* variable. The second is a parsed *<format specification>*. Set up the loop.
__str_format_seq:ff

```

16821 \cs_new:Npn \__str_format_seq:nn #1#2
16822 {
16823     \__str_format_seq_loop:nnNn { } {#2}
16824     #1
16825     { ? \__str_format_seq_end:w } { }

```

```

16826     }
16827 \cs_generate_variant:Nn \__str_format_seq:nn { ff }

```

(End of definition for __str_format_seq:nn.)

```

\__str_format_seq_loop:nnNn \__str_format_seq_loop:nnNn {<done>} {<parsed format>} \__seq_item:n
{<item>}

```

The first argument is the result of formatting the items read so far. The third argument is a single token (__seq_item:n), until we reach the end of the sequence, where \use_none:n #3 ends the loop.

```

16828 \cs_new:Npn \__str_format_seq_loop:nnNn #1#2#3#4
16829 {
16830     \use_none:n #3
16831     \exp_args:Nf \__str_format_seq_loop:nnNn
16832     { \use:nf {#1} { \__str_format_tl:NNNnnNn #2 {#4} } }
16833     {#2}
16834 }

```

(End of definition for __str_format_seq_loop:nnNn.)

__str_format_seq_end:w Pick the right piece in the loop above.

```

16835 \cs_new:Npn \__str_format_seq_end:w #1#2#3#4 { \use_ii:nnn #3 }

```

(End of definition for __str_format_seq_end:w.)

61.6 Formatting integers

\int_format:nn Evaluate the first argument and feed it to __str_format_int:nn.

```

16836 \cs_new:Npn \int_format:nn #1
16837 { \exp_args:Nf \__str_format_int:nn { \int_eval:n {#1} } }

```

(End of definition for \int_format:nn. This function is documented on page 182.)

__str_format_int:nn Parse the <format specification> and feed it to __str_format_int:NNNnnNn. Then convert the result to a string

```

16838 \cs_new:Npn \__str_format_int:nn #1#2
16839 {
16840     \tl_to_str:f
16841     {
16842         \exp_last_unbraced:Nf \__str_format_int:NNNnnNn
16843         { \__str_format_parse:n {#2} }
16844         {#1}
16845     }
16846 }

```

(End of definition for __str_format_int:nn.)

```

\__str_format_int:NNNnnNn \__str_format_int:NNNnnNn <fill> <alignment> <sign> {<width>} {<precision>}
<style> {<integer>}

```

First set the default alignment ? to >. Place the modified information after a trailing \s__str_stop for later retrieval. Then check the <sign>: if the integer is negative, always put -. Otherwise, if the format's <sign> is ~, put a space (with category "other"); if it is + put +; if it is - (default), put nothing, represented as a brace group. The width #4

will be useful later, store it after `\s__str_stop`. Afterwards, check that the `<precision>` was absent. Finally, dispatch depending on the `<style>`.

```

16847 \cs_new:Npn \__str_format_int:NNNnnNn #1#2#3#4#5#6#7
16848 {
16849   \token_if_eq_charcode:NNTF #2 ?
16850   { \__str_format_put:nw { #1 > } }
16851   { \__str_format_put:nw { #1 #2 } }
16852   \int_compare:nNnTF {#7} < 0
16853   { \__str_format_put:nw { - } }
16854   {
16855     \str_case:nnF {#3}
16856     {
16857       { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
16858       { + } { \__str_format_put:nw { + } }
16859     }
16860     { \__str_format_put:nw { { } } }
16861   }
16862   \__str_format_put:nw { {#4} }
16863   \tl_if_empty:nF {#5}
16864   {
16865     \msg_expandable_error:nnnn
16866     { str } { invalid-precision-format } {#5} {int}
16867   }
16868   \str_case:nnF {#6}
16869   {
16870     { ? } { \__str_format_int:NwnnNNn \use:n }
16871     { d } { \__str_format_int:NwnnNNn \use:n }
16872     { b } { \__str_format_int:NwnnNNn \int_to_bin:n }
16873     { o } { \__str_format_int:NwnnNNn \int_to_oct:n }
16874     { X } { \__str_format_int:NwnnNNn \int_to_Hex:n }
16875   }
16876   {
16877     \msg_expandable_error:nnnn
16878     { str } { invalid-style-format } {#6} { int }
16879     \__str_format_int:NwnnNNn \use:n
16880   }
16881   \s__str_stop {#7}
16882 }

```

(End of definition for `\__str_format_int:NNNnnNn`.)

```

\__str_format_int:NwnnNNn \__str_format_int:NwnnNNn <function> \s__str_stop {<width>} {<sign>}
<fill> <alignment> {<integer>}

```

Use the `format_align` function corresponding to the `<alignment>`, with the following arguments:

- the string formed by combining the sign `#4` with the result of converting the absolute value of the `<integer>` `#7` according to the conversion function `#1`;
- the `<width>`;
- the `<fill>` character.

```

16883 \cs_new:Npn \__str_format_int:NwnnNNn #1#2 \s__str_stop #3#4#5#6#7
16884 {

```

```

16885     \exp_args:Nc \exp_args:Nf
16886     { __str_format_align_#6:nnnN }
16887     { #1 { \int_abs:n {#7} } }
16888     {#4}
16889     {#3} #5
16890 }

```

(End of definition for `\__str_format_int:NwnnNNn`.)

61.7 Formatting floating points

`\fp_format:nn` Evaluate the first argument to a floating point number, and feed it to `\__str_format_fp:nn`. It would be more efficient to use internal floating point numbers but efficiency is not essential and the code is cleaner this way.

```

16891 \cs_new:Npn \fp_format:nn #1
16892 { \exp_args:Nf \__str_format_fp:nn { \fp_to_tl:n {#1} } }

```

(End of definition for `\fp_format:nn`. This function is documented on page 270.)

`\__str_format_fp:nn` Parse the *<format specification>* and feed it to `\__str_format_fp:NNNnnNn`. Then convert the result to a string

```

16893 \cs_new:Npn \__str_format_fp:nn #1#2
16894 {
16895     \tl_to_str:f
16896     {
16897         \exp_last_unbraced:Nf \__str_format_fp:NNNnnNn
16898         { \__str_format_parse:n {#2} }
16899         {#1}
16900     }
16901 }

```

(End of definition for `\__str_format_fp:nn`.)

`\__str_format_fp:NNNnnNn` `\__str_format_fp:NNNnnNn <fill> <alignment> <format sign> {<width>}`
`{<precision>} <style> {<floating point>}`

First set the default alignment ? to >. Place the modified information after a trailing `\s__str_stop` for later retrieval. Then check the *<format sign>* and the *<fp sign>*: if the floating point is negative, always put -. Otherwise (including nan), if the format's *<sign>* is ~, put a space (with category "other"); if it is + put +; if it is - (default), put nothing, represented as a brace group. The width #4 will be useful later, store it after `\s__str_stop`. Afterwards, check the *<precision>*: if it was not given, replace it by 6 (default precision) unless no *<style>* was given: in that case we want to use whatever precision is needed to fully describe the number. Finally, dispatch depending on the *<style>*.

```

16902 \cs_new:Npn \__str_format_fp:NNNnnNn
16903     #1#2#3#4#5#6#7
16904 {
16905     \token_if_eq_charcode:NNTF #2 ?
16906     { \__str_format_put:nw { #1 > } }
16907     { \__str_format_put:nw { #1 #2 } }
16908     \tl_if_head_eq_charcode:nNTF {#7} -
16909     { \__str_format_put:nw { - } }

```

```

16910     {
16911         \str_case:nnF {#3}
16912         {
16913             { ~ } { \__str_format_put:ow { \c_catcode_other_space_tl } }
16914             { + } { \__str_format_put:nw { + } }
16915         }
16916         { \__str_format_put:nw { { } } }
16917     }
16918     \__str_format_put:nw { {#4} }
16919     \tl_if_empty:nTF {#5}
16920     {
16921         \token_if_eq_meaning:NNTF #6 ?
16922         { \__str_format_put:nw { { } } }
16923         { \__str_format_put:nw { { 6 } } }
16924     }
16925     { \__str_format_put:nw { {#5} } }
16926     \str_case:nnF {#6}
16927     {
16928         { e } { \__str_format_fp:wnnnNNn \__str_format_fp_e:nn }
16929         { f } { \__str_format_fp:wnnnNNn \__str_format_fp_f:nn }
16930         { g } { \__str_format_fp:wnnnNNn \__str_format_fp_g:nn }
16931         { ? } { \__str_format_fp:wnnnNNn \__str_format_fp_g:nn }
16932     }
16933     {
16934         \msg_expandable_error:nnnn
16935         { str } { invalid-style-format } {#6} { fp }
16936         \__str_format_fp:wnnnNNn \__str_format_fp_g:nn
16937     }
16938     \s__str_stop {#7}
16939 }

```

(End of definition for __str_format_fp:NNnnNNn.)

```

\__str_format_fp:wnnnNNn      \__str_format_fp:wnnnNNn <formatting function> \s__str_stop {<precision>}
                              {<width>} {<sign>} <fill> <alignment> {<floating point>}

16940 \cs_new:Npn \__str_format_fp:wnnnNNn
16941       #1 \s__str_stop #2 #3 #4 #5#6 #7
16942     {
16943       \exp_args:Nc \exp_args:Nf
16944       { \__str_format_align_#6:nnnN }
16945       { #1 {#7} {#2} }
16946       {#4}
16947       {#3} #5
16948     }

```

(End of definition for __str_format_fp:wnnnNNn.)

__str_format_fp_round:nn Round the given floating point and take the absolute value (in this order, to play nicely with unusual rounding modes if we ever implement these).

```

16949 \cs_new:Npn \__str_format_fp_round:nn #1 #2
16950   { \fp_to_tl:n { abs ( round ( #1 , #2 - logb(#1) - 1 ) ) } }

```

(End of definition for __str_format_fp_round:nn.)

```

    \__str_format_fp_e:nn
    \__str_format_fp_e_aux:nn
    \__str_format_fp_e_aux:wnn

```

With the **e** type, first round to #4+1 significant figures (one before the decimal separator, #4 after), then filter out special cases, then convert to scientific notations. This order is important because rounding can produce infinities or zeros and because `\fp_to_scientific:n` does not accept nan nor inf.

```

16951 \cs_new:Npn \__str_format_fp_e:nn #1#2
16952 {
16953     \exp_args:Nf \__str_format_fp_e_aux:nn
16954     { \__str_format_fp_round:nn {#1} { #2 + 1 } }
16955     {#2}
16956 }
16957 \cs_new:Npn \__str_format_fp_e_aux:nn #1#2
16958 {
16959     \str_case:nnF {#1}
16960     {
16961         { inf } { inf }
16962         { nan } { nan }
16963     }
16964     {
16965         \exp_last_unbraced:Nf \__str_format_fp_e_aux:wnn
16966         { \fp_to_scientific:n {#1} } ;
16967         {#2}
16968     }
16969 }
16970 \cs_new:Npn \__str_format_fp_e_aux:wnn #1 . #2 e #3 ; #4
16971 {
16972     \__str_format_put:nw { e #3 }
16973     \int_compare:nNnTF {#4} < \c__fp_prec_int
16974     {
16975         \__str_format_put:fw { \str_range:nnn { #2 } { 1 } { #4 } }
16976         \__str_format_put:nw { #1 . }
16977     }
16978     {
16979         \__str_format_put:fw
16980         { \prg_replicate:nn { #4 - \c__fp_prec_int + 1 } { 0 } }
16981         \__str_format_put:nw { #1 . #2 }
16982     }
16983     \use_none:n \s__str_stop
16984 }

```

(End of definition for `\__str_format_fp_e:nn`, `\__str_format_fp_e_aux:nn`, and `\__str_format_fp_e_aux:wnn`.)

```

    \__str_format_fp_f:nn
    \__str_format_fp_f_aux:wwn

```

With the **f** type, first round to #4 (absolute) decimal places then filter out special cases, then in normal cases pad with zeros.

```

16985 \cs_new:Npn \__str_format_fp_f:nn #1#2
16986 {
16987     \exp_args:Nf \__str_format_fp_f_aux:nn
16988     { \fp_to_tl:n { abs ( round ( #1 , #2 ) ) } }
16989     {#2}
16990 }
16991 \cs_new:Npn \__str_format_fp_f_aux:nn #1#2
16992 {
16993     \str_case:nnF {#1}
16994     {

```

```

16995         { inf } { inf }
16996         { nan } { nan }
16997     }
16998     {
16999         \exp_last_unbraced:Nf \__str_format_fp_f_aux:wwwn
17000         { \fp_to_decimal:n {#1} } . . ; {#2}
17001     }
17002 }
17003 \cs_new:Npn \__str_format_fp_f_aux:wwwn #1 . #2 . #3 ; #4
17004 {
17005     \use:nf { #1 . #2 }
17006     { \prg_replicate:nn { #4 - \__str_count:n {#2} } {0} }
17007 }

```

(End of definition for `\__str_format_fp_f:nn` and `\__str_format_fp_f_aux:wwwn`.)

```

\__str_format_fp_g:nn
\__str_format_fp_g_aux:wn
\__str_format_fp_to_scientific:n
\__str_format_fp_trim:w

```

With the `g` type, a special case is when `#2` is empty (no style nor precision in the original specification): then we output the number without rounding (and without its sign). Otherwise round to `#2` significant figures before filtering out special cases. (A `<precision>` of 0 is treated like a precision of 1.) Distinguish exponents $-4 \leq \langle \textit{exponent} \rangle < \langle \textit{precision} \rangle$ from others and use essentially the `f` or `e` presentations in these two cases, but trimming trailing zeros. Because we don't need to keep a fixed number of digits after the decimal point we can simply use `\fp_to_decimal:n` and `\fp_to_scientific:n`, and in the second case post-process the result by trimming zeros and a period.

```

17008 \cs_new:Npn \__str_format_fp_g:nn #1#2
17009 {
17010     \tl_if_empty:nTF {#2} { \fp_to_tl:n { abs(#1) } }
17011     {
17012         \exp_args:Nf \__str_format_fp_g_aux:nn
17013         { \__str_format_fp_round:nn {#1} { \int_max:nn {1} {#2} } }
17014         { \int_max:nn {1} {#2} }
17015     }
17016 }
17017 \cs_new:Npn \__str_format_fp_g_aux:nn #1#2
17018 {
17019     \str_case:nnF {#1}
17020     {
17021         { 0 } { 0 }
17022         { inf } { inf }
17023         { nan } { nan }
17024     }
17025     {
17026         \fp_compare:nTF { 1e-4 <= #1 < 1e#2 }
17027         { \fp_to_decimal:n {#1} }
17028         {
17029             \exp_last_unbraced:Nf \__str_format_fp_trim:w
17030             { \fp_to_scientific:n {#1} }
17031         }
17032     }
17033 }

```

(End of definition for `\__str_format_fp_g:nn` and others.)

```

\__str_format_fp_trim:w
\__str_format_fp_trim_loop:w
\__str_format_fp_trim_dot:w
\__str_format_fp_trim_end:w

```

If `#1` ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing

zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

17034 \cs_new:Npn \__str_format_fp_trim:w #1 e
17035 {
17036   \__str_format_fp_trim_loop:w #1
17037   ; \__str_format_fp_trim_loop:w 0; \__str_format_fp_trim_dot:w .; \s__str_stop e
17038 }
17039 \cs_new:Npn \__str_format_fp_trim_loop:w #1 0; #2 { #2 #1 ; #2 }
17040 \cs_new:Npn \__str_format_fp_trim_dot:w #1 .; { \__str_format_fp_trim_end:w #1 ; }
17041 \cs_new:Npn \__str_format_fp_trim_end:w #1 ; #2 \s__str_stop { #1 }

```

(End of definition for __str_format_fp_trim:w and others.)

61.8 Messages

All of the messages are produced expandably, so there is no need for an extra-text.

```

17042 \msg_new:nnn { str } { invalid-format }
17043 { Invalid-format~'#1'. }
17044 \msg_new:nnn { str } { invalid-align-format }
17045 { Invalid-alignment~'#1'~for~type~'#2'. }
17046 \msg_new:nnn { str } { invalid-sign-format }
17047 { Invalid-sign~'#1'~for~type~'#2'. }
17048 \msg_new:nnn { str } { invalid-precision-format }
17049 { Invalid-precision~'#1'~for~type~'#2'. }
17050 \msg_new:nnn { str } { invalid-style-format }
17051 { Invalid-style~'#1'~for~type~'#2'. }
17052 </code>

```

Chapter 62

l3quark implementation

The following test files are used for this code: *m3quark001.lvt*.

17053 `\*code`

62.1 Quarks

17054 `\@@=quark`

`\quark_new:N` Allocate a new quark.

17055 `\cs_new_protected:Npn \quark_new:N #1`
17056 `{`
17057 `\__kernel_chk_if_free_cs:N #1`
17058 `\cs_gset_nopar:Npn #1 {#1}`
17059 `}`

(End of definition for `\quark_new:N`. This function is documented on page 152.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

`\q_mark`
`\q_no_value`
`\q_stop`

17060 `\quark_new:N \q_nil`
17061 `\quark_new:N \q_mark`
17062 `\quark_new:N \q_no_value`
17063 `\quark_new:N \q_stop`

(End of definition for `\q_nil` and others. These variables are documented on page 152.)

`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to
`\q_recursion_stop` whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

17064 `\quark_new:N \q_recursion_tail`
17065 `\quark_new:N \q_recursion_stop`

(End of definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 153.)

`\s__quark` Private scan mark used in l3quark. We don’t have l3scan yet, so we declare the scan mark here and add it to the scan mark pool later.

17066 `\cs_new_eq:NN \s__quark \scan_stop:`

(End of definition for `\s__quark`.)

`\q__quark_nil` Private quark use for some tests.

```
17067 \quark_new:N \q__quark_nil
```

(End of definition for `\q__quark_nil`.)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
17068 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
17069 {
17070   \if_meaning:w \q_recursion_tail #1
17071   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
17072   \fi:
17073 }
17074 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
17075 {
17076   \if_meaning:w \q_recursion_tail #1
17077   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
17078   \else:
17079   \exp_after:wN \use_none:n
17080   \fi:
17081 }
```

(End of definition for `\quark_if_recursion_tail_stop:N` and `\quark_if_recursion_tail_stop_do:Nn`. These functions are documented on page 153.)

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`
`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:nn`
`\__quark_if_recursion_tail:w`

See `\quark_if_nil:nTF` for the details. Expanding `\__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if #1 is exactly `\q_recursion_tail`.

```
17082 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
17083 {
17084   \tl_if_empty:oTF
17085   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
17086   { \use_none_delimit_by_q_recursion_stop:w }
17087   { }
17088 }
17089 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
17090 {
17091   \tl_if_empty:oTF
17092   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
17093   { \use_i_delimit_by_q_recursion_stop:nw }
17094   { \use_none:n }
17095 }
17096 \cs_new:Npn \__quark_if_recursion_tail:w
17097   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
17098 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
17099 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(End of definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `\__quark_if_recursion_tail:w`. These functions are documented on page 153.)

`\quark_if_recursion_tail_break:NN` Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping
`\quark_if_recursion_tail_break:nN` using #2.

```

17100 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
17101 {
17102   \if_meaning:w \q_recursion_tail #1
17103   \exp_after:wN #2
17104   \fi:
17105 }
17106 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
17107 {
17108   \tl_if_empty:oT
17109   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
17110   {#2}
17111 }

```

(End of definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 153.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_nil:NTF` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value_p:N` wrongly given a string like aabc instead of a single token.¹⁰

```

17112 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T, F, TF }
17113 {
17114   \if_meaning:w \q_nil #1
17115   \prg_return_true:
17116   \else:
17117   \prg_return_false:
17118   \fi:
17119 }
17120 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T, F, TF }
17121 {
17122   \if_meaning:w \q_no_value #1
17123   \prg_return_true:
17124   \else:
17125   \prg_return_false:
17126   \fi:
17127 }
17128 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
17129 { c } { p, T, F, TF }

```

(End of definition for `\quark_if_nil:N` and `\quark_if_no_value:N`. These functions are documented on page 152.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:nTF`. Expanding `\__quark_if_nil:w` once is safe thanks
`\quark_if_nil_p:V` to the trailing `\q_nil ???`. The result of expanding once is empty if and only if both
`\quark_if_nil_p:o` delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens ?!.
`\quark_if_nil:nTF` Thanks to the leading {}, the argument #1 is empty if and only if the argument of
`\quark_if_nil:VTF` `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_-`
`\quark_if_nil:oTF` `nil` is followed immediately by ? or by {}?, coming either from the trailing tokens in the
`\quark_if_no_value_p:n` definition of `\quark_if_nil:n`, or from its argument. In the first case, `\__quark_if_-`
`\quark_if_no_value:nTF` `nil:w` is followed by `\q_nil {}? !\q_nil ???`, hence #3 is delimited by the final ?!,
`\__quark_if_nil:w` and the test returns true as wanted. In the second case, the result is not empty since

¹⁰It may still loop in special circumstances however!

the first ?! in the definition of `\quark_if_nil:n` stop #3. The auxiliary here is the same as `\__tl_if_empty_if:o`, with the same comments applying.

```

17130 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
17131 {
17132   \__quark_if_empty_if:o
17133   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
17134   \prg_return_true:
17135   \else:
17136   \prg_return_false:
17137   \fi:
17138 }
17139 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
17140 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
17141 {
17142   \__quark_if_empty_if:o
17143   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
17144   \prg_return_true:
17145   \else:
17146   \prg_return_false:
17147   \fi:
17148 }
17149 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
17150 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
17151 { V , o } { p , TF , T , F }
17152 \cs_new:Npn \__quark_if_empty_if:o #1
17153 {
17154   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
17155   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
17156 }

```

(End of definition for `\quark_if_nil:nTF` and others. These functions are documented on page 152.)

`\__kernel_quark_new_test:N` The function `\__kernel_quark_new_test:N` defines #1 in a similar way as `\quark_if_recursion_tail...` functions (as described below), using `\q_<namespace>_recursion_tail` as the test quark and `\q_<namespace>_recursion_stop` as the delimiter quark, where the `<namespace>` is determined as the first `_`-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

`:n` gives an analogue of `\quark_if_recursion_tail_stop:n`
`:nn` gives an analogue of `\quark_if_recursion_tail_stop_do:nn`
`:nN` gives an analogue of `\quark_if_recursion_tail_break:nN`
`:N` gives an analogue of `\quark_if_recursion_tail_stop:N`
`:Nn` gives an analogue of `\quark_if_recursion_tail_stop_do:Nn`
`:NN` gives an analogue of `\quark_if_recursion_tail_break:NN`

Any other signature causes an error, as does a function without signature.

_kernel_quark_new_conditional:Nn

Similar to _kernel_quark_new_test:N, but defines quark branching conditionals like \quark_if_nil:nTF that test for the quark \q_<namespace>_<name>. The <namespace> and <name> are determined from the conditional #1, which must take the rather rigid form _<namespace>_quark_if_<name>:<arg spec>. There are only two cases for the <arg spec> here:

:n gives an analogue of \quark_if_nil:nTF

:N gives an analogue of \quark_if_nil:NTF

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as l3tl is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple \if_meaning:w \q_nil <string> \q_nil suffices.

```

17157 \cs_new_protected:Npn \_kernel\_quark\_new\_test:N #1
17158 { \_quark\_new\_test\_aux:Ne #1 { \_quark\_module\_name:N #1 } }
\_quark\_new\_test:NNNn 17159 \cs_new_protected:Npn \_quark\_new\_test\_aux:Nn #1 #2
\_quark\_new\_test:Nccn 17160 {
\_quark\_new\_test\_aux:nnNNnnnn 17161 \if\_meaning:w \q\_nil #2 \q\_nil
\_quark\_new\_conditional:Nnnn 17162 \msg\_error:nne { quark } { invalid-function }
\_quark\_new\_conditional:Neen 17163 { \token\_to\_str:N #1 }
17164 \else:
17165 \_quark\_new\_test:Nccn #1
17166 { q\_#2\_recursion\_tail } { q\_#2\_recursion\_stop } { \_#2 }
17167 \fi:
17168 }
17169 \cs\_generate\_variant:Nn \_quark\_new\_test\_aux:Nn { Ne }
17170 \cs\_new\_protected:Npn \_quark\_new\_test:NNNn #1
17171 {
17172 \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
17173 { \cs\_split\_function:N #1 }
17174 #1 { test }
17175 }
17176 \cs\_generate\_variant:Nn \_quark\_new\_test:NNNn { Ncc }
17177 \cs\_new\_protected:Npn \_kernel\_quark\_new\_conditional:Nn #1
17178 {
17179 \_quark\_new\_conditional:Neen #1
17180 { \_quark\_quark\_conditional\_name:N #1 }
17181 { \_quark\_module\_name:N #1 }
17182 }
17183 \cs\_new\_protected:Npn \_quark\_new\_conditional:Nnnn #1#2#3#4
17184 {
17185 \if\_meaning:w \q\_nil #2 \q\_nil
17186 \msg\_error:nne { quark } { invalid-function }
17187 { \token\_to\_str:N #1 }
17188 \else:
17189 \if\_meaning:w \q\_nil #3 \q\_nil
17190 \msg\_error:nne { quark } { invalid-function }
17191 { \token\_to\_str:N #1 }
17192 \else:
17193 \exp\_last\_unbraced:Nf \_quark\_new\_test\_aux:nnNNnnnn
17194 { \cs\_split\_function:N #1 }

```

```

17195         #1 { conditional }
17196         {#2} {#3} {#4}
17197     \fi:
17198 \fi:
17199 }
17200 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nee }
17201 \cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
17202 {
17203     \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
17204     {
17205         \msg_error:nnee { quark } { invalid-function }
17206         { \token_to_str:N #4 } {#2}
17207         \use_none:nnn
17208     }
17209 }

```

(End of definition for __kernel_quark_new_test:N and others.)

__quark_new_test_n:Nnnn These macros implement the six possibilities mentioned above, passing the right arguments to __quark_new_test_aux_do:nNNnnnnNNn, which defines some auxiliaries, and then to __quark_new_test_define_tl:nNnNNn (:n(n) variants) or to __quark_new_test_define_ifx:nNnNNn (:N(n)) which define the main conditionals.

```

17210 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
17211 {
17212     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
17213     \__quark_new_test_define_tl:nNnNNn #1 { }
17214 }
17215 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
17216 {
17217     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
17218     \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
17219 }
17220 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
17221 {
17222     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
17223     \__quark_new_test_define_break_tl:nNNNNn #1 { }
17224 }
17225 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
17226 {
17227     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
17228     \__quark_new_test_define_ifx:nNnNNn #1 { }
17229 }
17230 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
17231 {
17232     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
17233     \__quark_new_test_define_ifx:nNnNNn #1
17234     { \else: \exp_after:wN \use_none:n }
17235 }
17236 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
17237 {
17238     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
17239     \__quark_new_test_define_break_ifx:nNNNNn #1 { }
17240 }

```

(End of definition for __quark_new_test_n:Nnnn and others.)

_quark_new_test_aux_do:nNNnnnnNNn
_quark_test_define_aux:NNNNnnNNn

_quark_new_test_aux_do:nNNnnnnNNn makes the control sequence names which will be used by _quark_test_define_aux:NNNNnnNNn, and then later by _quark_new_test_define_tl:nNnNNn or _quark_new_test_define_ifx:nNnNNn. The control sequences defined here are analogous to _quark_if_recursion_tail:w and to \use_-(none|i)_delimit_by_q_recursion_stop:(|n)w.

The name is composed by the name-space and the name of the quarks. Suppose _kernel_quark_new_test:N was used with:

_kernel_quark_new_test:N _test_quark_tail:n

then the first auxiliary will be _test_quark_recursion_tail:w, and the second one will be _test_use_none_delimit_by_q_recursion_stop:w.

Note that the actual quarks are *not* defined here. They should be defined separately using \quark_new:N.

```

17241 \cs_new_protected:Npn \_quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
17242 {
17243   \exp_args:Ncc \_quark_test_define_aux:NNNNnnNNn
17244   { #1 \_quark_recursion_tail:w }
17245   { #1 \_use_ #4 \_delimit_by_q_recursion_stop: #5 w }
17246   #2 #3
17247 }
17248 \cs_new_protected:Npn \_quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
17249 {
17250   \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
17251   \cs_gset:Npn #2 ##1 #6 #4 {#5}
17252   #7 {##1} #1 #2 #3
17253 }
```

(End of definition for _quark_new_test_aux_do:nNNnnnnNNn and _quark_test_define_aux:NNNNnnNNn.)

_quark_new_test_define_tl:nNnNNn
_quark_new_test_define_ifx:nNnNNn
_quark_new_test_define_break_tl:nNNNNn
_quark_new_test_define_break_ifx:nNNNNn

Finally, these two macros define the main conditional function using what's been set up before.

```

17254 \cs_new_protected:Npn \_quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
17255 {
17256   \cs_new:Npn #5 #1
17257   {
17258     \tl_if_empty:oTF
17259     { #2 {} ##1 {} ?! #4 ??! }
17260     {#3} {#6}
17261   }
17262 }
17263 \cs_new_protected:Npn \_quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
17264 {
17265   \cs_new:Npn #5 #1
17266   {
17267     \if_meaning:w #4 ##1
17268     \exp_after:wN #3
17269     #6
17270     \fi:
17271   }
17272 }
17273 \cs_new_protected:Npn \_quark_new_test_define_break_tl:nNNNNn #1 #2 #3
17274 { \_quark_new_test_define_tl:nNnNNn {##1##2} #2 {##2} }
17275 \cs_new_protected:Npn \_quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
```

```
17276 { \_quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }
```

(End of definition for _quark_new_test_define_tl:nNnNNn and others.)

```
\_quark_new_conditional_n:Nnnn
\_quark_new_conditional_N:Nnnn
\_quark_new_conditional_n_aux:NNNn
\_quark_new_conditional_N_aux:NNNn
```

These macros implement the two possibilities for branching quark conditionals. To avoid constructing without defining the _<type>_if_quark_<name>:w helper, N-type function accepts a \prg_do_nothing: as a placeholder.

```
17277 \cs_new_protected:Npn \_quark_new_conditional_n:Nnnn #1 #2 #3
17278 {
17279   \exp_args:Ncc \_quark_new_conditional_n_aux:NNNn
17280   { \_ #3 _if_quark_ #2 :w } { q\_ #3 _ #2 } #1
17281 }
17282 \cs_new_protected:Npn \_quark_new_conditional_N:Nnnn #1 #2 #3
17283 {
17284   \exp_args:NNc \_quark_new_conditional_N_aux:NNNn
17285   \prg_do_nothing: { q\_ #3 _ #2 } #1
17286 }
17287 \cs_new_protected:Npn \_quark_new_conditional_n_aux:NNNn #1 #2 #3 #4
17288 {
17289   \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1##2 }
17290   \prg_new_conditional:Npnn #3 ##1 {#4}
17291   {
17292     \_quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??? }
17293     \prg_return_true:
17294   }
17295   \else:
17296     \prg_return_false:
17297   \fi:
17298 }
17299 \cs_new_protected:Npn \_quark_new_conditional_N_aux:NNNn #1 #2 #3 #4
17300 {
17301   \prg_new_conditional:Npnn #3 ##1 {#4}
17302   {
17303     \if_meaning:w #2 ##1
17304     \prg_return_true:
17305   }
17306   \else:
17307     \prg_return_false:
17308   \fi:
17309 }
```

(End of definition for _quark_new_conditional_n:Nnnn and others.)

```
\_quark_module_name:N
\_quark_module_name:w
\_quark_module_name_loop:w
\_quark_module_name_end:w
```

_quark_module_name:N takes a control sequence and returns its <module> name, determined as the first non-empty non-single-character word, separated by _ or :. These rules give the correct result for public functions \<module>_..., private functions _<module>_..., and variables such as \l_<module>_.... If no valid module is found the result is an empty string. The approach is to first cut off everything after the (first) : if any is present, then repeatedly grab _-delimited words until finding one of length at least 2 (we use low-level tests as l3tl is not fully available when _kernel_quark_new_test:N is first used. If no <module> is found (such as in \::n) we get the trailing marker \use_none:n {}, which expands to nothing.

```
17310 \cs_set:Npn \_quark_tmp:w #1#2
17311 {
```

```

17312 \cs_new:Npn \__quark_module_name:N ##1
17313 {
17314     \exp_last_unbraced:Nf \__quark_module_name:w
17315     { \cs_to_str:N ##1 } #1 \s__quark
17316 }
17317 \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark
17318 { \__quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
17319 \cs_new:Npn \__quark_module_name_loop:w ##1 #2
17320 {
17321     \use_i_ii:nnn \if_meaning:w \prg_do_nothing:
17322     ##1 \prg_do_nothing: \prg_do_nothing:
17323     \exp_after:wN \__quark_module_name_loop:w
17324     \else:
17325     \__quark_module_name_end:w ##1
17326     \fi:
17327 }
17328 \cs_new:Npn \__quark_module_name_end:w
17329 ##1 \fi: ##2 \s__quark { \fi: ##1 }
17330 }
17331 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }

```

(End of definition for __quark_module_name:N and others.)

__quark_quark_conditional_name:N determines the quark name that the quark conditional function ##1 queries, as the part of the function name between `_quark_if_` and the trailing `:`. Again we define it through `\__quark_tmp:w`, which receives `:` as #1 and `_quark_if_` as #2. The auxiliary `\__quark_quark_conditional_name:w` returns the part between the first `_quark_if_` and the next `:`, and we apply this auxiliary to the function name followed by `:` (in case the function name is lacking a signature), and `_quark_if_:` so that `\__quark_quark_conditional_name:N` returns an empty string if `_quark_if_` is not present.

```

17332 \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
17333 {
17334     \cs_new:Npn \__quark_quark_conditional_name:N ##1
17335     {
17336         \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
17337         { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
17338     }
17339     \cs_new:Npn \__quark_quark_conditional_name:w
17340     ##1 #2 ##2 #1 ##3 \s__quark {##2}
17341 }
17342 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark

```

(End of definition for __quark_quark_conditional_name:N and __quark_quark_conditional_name:w.)

62.2 Scan marks

```

17343 <@@=scan>

```

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```

17344 \cs_new_protected:Npn \scan_new:N #1
17345 {

```

```

17346 \tl_if_in:NnTF \g__scan_marks_tl { #1 }
17347 {
17348   \msg_error:nne { scanmark } { already-defined }
17349   { \token_to_str:N #1 }
17350 }
17351 {
17352   \tl_gput_right:Nn \g__scan_marks_tl {#1}
17353   \cs_new_eq:NN #1 \scan_stop:
17354 }
17355 }

```

(End of definition for `\scan_new:N`. This function is documented on page 155.)

`\s_stop` We only declare one scan mark here, more can be defined by specific modules. Can't use `\scan_new:N` yet because `l3tl` isn't loaded, so define `\s_stop` by hand and add it to `\g__scan_marks_tl`. We also add the scan marks declared earlier to the pool here. Since they live in a different namespace, a little `DocStrip` cheating is necessary.

```

17356 \cs_new_eq:NN \s_stop \scan_stop:
17357 \cs_gset_nopar:Npn \g__scan_marks_tl
17358 {
17359   \s_stop
17360   <@@=quark>
17361   \s__quark
17362   <@@=cs>
17363   \s__cs_mark
17364   \s__cs_stop
17365   <@@=scan>
17366 }

```

(End of definition for `\s_stop` and `\g__scan_marks_tl`. This variable is documented on page 155.)

`\use_none_delimit_by_s_stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

```

17367 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }

```

(End of definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 155.)

```

17368 </code>

```


Chapter 63

l3seq implementation

The following test files are used for this code: *m3seq002,m3seq003*.

17369 $\langle *code \rangle$

17370 $\langle @@=seq \rangle$

A sequence is a control sequence whose top-level expansion is of the form “ $\backslash s\_seq \backslash\_seq\_item:n \{ \langle item_1 \rangle \} \dots \backslash\_seq\_item:n \{ \langle item_n \rangle \}$ ”, with a leading scan mark followed by n items of the same form. An earlier implementation used the structure “ $\backslash seq\_elt:w \langle item_1 \rangle \backslash seq\_elt\_end: \dots \backslash seq\_elt:w \langle item_n \rangle \backslash seq\_elt\_end:$ ”. This allowed rapid searching using a delimited function, but was not suitable for items containing $\{, \}$ and $\#$ tokens, and also lead to the loss of surrounding braces around items

 $\backslash\_seq\_item:n \star \backslash\_seq\_item:n \{ \langle item \rangle \}$

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

 $\backslash\_seq\_push\_item\_def:n \backslash\_seq\_push\_item\_def:n \{ \langle code \rangle \}$

 $\backslash\_seq\_push\_item\_def:e$ Saves the definition of $\backslash\_seq\_item:n$ and redefines it to accept one parameter and expand to $\langle code \rangle$. This function should always be balanced by use of $\backslash\_seq\_pop\_item\_def:.$

 $\backslash\_seq\_pop\_item\_def: \backslash\_seq\_pop\_item\_def:$

Restores the definition of $\backslash\_seq\_item:n$ most recently saved by $\backslash\_seq\_push\_item\_def:n$. This function should always be used in a balanced pair with $\backslash\_seq\_push\_item\_def:n$.

 $\backslash s\_seq$ This private scan mark.

17371 $\backslash scan\_new:N \backslash s\_seq$

(End of definition for $\backslash s\_seq$.)

 $\backslash s\_seq\_mark$ Private scan marks. $\backslash s\_seq\_stop$ 17372 $\backslash scan\_new:N \backslash s\_seq\_mark$

17373 $\backslash scan\_new:N \backslash s\_seq\_stop$

(End of definition for $\backslash s\_seq\_mark$ and $\backslash s\_seq\_stop$.)

`\__seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
17374 \cs_new:Npn \__seq_item:n
17375 {
17376     \msg_expandable_error:nn { seq } { misused }
17377     \use_none:n
17378 }
```

(End of definition for __seq_item:n.)

`\l__seq_tmpa_tl` Scratch space for various internal uses.

```
\l__seq_tmpb_tl
17379 \tl_new:N \l__seq_tmpa_tl
17380 \tl_new:N \l__seq_tmpb_tl
```

(End of definition for \l__seq_tmpa_tl and \l__seq_tmpb_tl.)

`\__seq_tmp:w` Scratch function for internal use.

```
17381 \cs_new_eq:NN \__seq_tmp:w ?
```

(End of definition for __seq_tmp:w.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.

```
17382 \tl_const:Nn \c_empty_seq { \s_seq }
```

(End of definition for \c_empty_seq. This variable is documented on page 169.)

63.1 Allocation and initialization

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.

```
\seq_new:c
17383 \cs_new_protected:Npn \seq_new:N #1
17384 {
17385     \__kernel_chk_if_free_cs:N #1
17386     \cs_gset_eq:NN #1 \c_empty_seq
17387 }
17388 \cs_generate_variant:Nn \seq_new:N { c }
```

(End of definition for \seq_new:N. This function is documented on page 156.)

`\seq_clear:N` Clearing a sequence is similar to setting it equal to the empty one.

```
\seq_clear:c
17389 \cs_new_protected:Npn \seq_clear:N #1
\seq_gclear:N
17390 { \seq_set_eq:NN #1 \c_empty_seq }
\seq_gclear:c
17391 \cs_generate_variant:Nn \seq_clear:N { c }
17392 \cs_new_protected:Npn \seq_gclear:N #1
17393 { \seq_gset_eq:NN #1 \c_empty_seq }
17394 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(End of definition for \seq_clear:N and \seq_gclear:N. These functions are documented on page 156.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

```
\seq_clear_new:c
17395 \cs_new_protected:Npn \seq_clear_new:N #1
\seq_gclear_new:N
17396 { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\seq_gclear_new:c
17397 \cs_generate_variant:Nn \seq_clear_new:N { c }
17398 \cs_new_protected:Npn \seq_gclear_new:N #1
17399 { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
17400 \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(End of definition for `\seq_clear_new:N` and `\seq_gclear_new:N`. These functions are documented on page 156.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.
`\seq_set_eq:cN`
`\seq_set_eq:Nc`
`\seq_set_eq:cc`
`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

(End of definition for `\seq_set_eq:NN` and `\seq_gset_eq:NN`. These functions are documented on page 156.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.
`\seq_set_from_clist:cN`
`\seq_set_from_clist:Nc`
`\seq_set_from_clist:cc`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:cN`
`\seq_gset_from_clist:Nc`
`\seq_gset_from_clist:cc`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

(End of definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 157.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.
`\seq_const_from_clist:cn`

(End of definition for `\seq_const_from_clist:Nn`. This function is documented on page 157.)

```

\seq_set_split:Nnn When the separator is empty, everything is very simple, just map \__seq_wrap_item:n
\seq_set_split:NVn through the items of the last argument. For non-trivial separators, the goal is to split a
\seq_set_split:NnV given token list at the marker, strip spaces from each item, and remove one set of outer
\seq_set_split:NVV braces if after removing leading and trailing spaces the item is enclosed within braces. Af-
\seq_set_split:Nne ter \tl_replace_all:Nnn, the token list \l__seq_tmpa_tl is a repetition of the pattern
\seq_set_split:Nee \__seq_set_split:Nw \prg_do_nothing: <item with spaces> \__seq_set_split_-
\seq_set_split:Nnx end:. Then, x-expansion causes \__seq_set_split:Nw to trim spaces, and leaves its
\seq_set_split:Nxx result as \__seq_set_split:w <trimmed item> \__seq_set_split_end:. This is then
\seq_gset_split:Nnn converted to the l3seq internal structure by another x-expansion. In the first step, we
\seq_gset_split:NVn insert \prg_do_nothing: to avoid losing braces too early: that would cause space trim-
\seq_gset_split:NnV ming to act within those lost braces. The second step is solely there to strip braces which
\seq_gset_split:NVV are outermost after space trimming.
\seq_gset_split:Nne
\seq_gset_split:Nee
\seq_gset_split:Nnx
\seq_gset_split:Nxx
\seq_set_split_keep_spaces:Nnn
\seq_set_split_keep_spaces:NnV
\seq_gset_split_keep_spaces:Nnn
\seq_gset_split_keep_spaces:NnV
\__seq_set_split:NNnn
  \__seq_set_split:Nw
  \__seq_set_split:w
\__seq_set_split_end:
17441 \cs_new_protected:Npn \seq_set_split:Nnn
17442   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \tl_trim_spaces:n }
17443 \cs_new_protected:Npn \seq_gset_split:Nnn
17444   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \tl_trim_spaces:n }
17445 \cs_new_protected:Npn \seq_set_split_keep_spaces:Nnn
17446   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \exp_not:n }
17447 \cs_new_protected:Npn \seq_gset_split_keep_spaces:Nnn
17448   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \exp_not:n }
17449 \cs_new_protected:Npn \__seq_set_split:NNNnn #1#2#3#4#5
17450   {
17451     \tl_if_empty:nTF {#4}
17452     {
17453       \tl_set:Nn \l__seq_tmpa_tl
17454         { \tl_map_function:nN {#5} \__seq_wrap_item:n }
17455     }
17456     {
17457       \tl_set:Nn \l__seq_tmpa_tl
17458       {
17459         \__seq_set_split:Nw #2 \prg_do_nothing:
17460         #5
17461         \__seq_set_split_end:
17462       }
17463       \tl_replace_all:Nnn \l__seq_tmpa_tl {#4}
17464       {
17465         \__seq_set_split_end:
17466         \__seq_set_split:Nw #2 \prg_do_nothing:
17467       }
17468       \__kernel_tl_set:Nx \l__seq_tmpa_tl { \l__seq_tmpa_tl }
17469     }
17470     #1 #3 { \s_seq \l__seq_tmpa_tl }
17471   }
17472 \cs_new:Npn \__seq_set_split:Nw #1#2 \__seq_set_split_end:
17473   {
17474     \exp_not:N \__seq_set_split:w
17475     \exp_args:No #1 {#2}
17476     \exp_not:N \__seq_set_split_end:
17477   }
17478 \cs_new:Npn \__seq_set_split:w #1 \__seq_set_split_end:
17479   { \__seq_wrap_item:n {#1} }

```

```

17480 \cs_generate_variant:Nn \seq_set_split:Nnn { NV , NnV , NVV , Nne , Nee }
17481 \cs_generate_variant:Nn \seq_set_split:Nnn { Nnx , Nxx }
17482 \cs_generate_variant:Nn \seq_gset_split:Nnn { NV , NnV , NVV , Nne , Nee }
17483 \cs_generate_variant:Nn \seq_gset_split:Nnn { Nnx , Nxx }
17484 \cs_generate_variant:Nn \seq_set_split_keep_spaces:Nnn { NnV }
17485 \cs_generate_variant:Nn \seq_gset_split_keep_spaces:Nnn { NnV }

```

(End of definition for `\seq_set_split:Nnn` and others. These functions are documented on page 157.)

`\seq_set_filter:NNn` Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `\__seq_wrap_item:n` function inserts the relevant `\__seq_item:n` without expansion in the input stream, hence in the x-expanding assignment.

```

17486 \cs_new_protected:Npn \seq_set_filter:NNn
17487 { \__seq_set_filter:NNNn \__kernel_tl_set:Nx }
17488 \cs_new_protected:Npn \seq_gset_filter:NNn
17489 { \__seq_set_filter:NNNn \__kernel_tl_gset:Nx }
17490 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
17491 {
17492   \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
17493   #1 #2 { #3 }
17494   \__seq_pop_item_def:
17495 }

```

(End of definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `\__seq_set_filter:NNNn`. These functions are documented on page 158.)

```

\seq_set_regex_extract_once:Nnn
\seq_set_regex_extract_once:cnn
\seq_gset_regex_extract_once:Nnn
\seq_gset_regex_extract_once:cnn
\seq_set_regex_extract_all:Nnn
\seq_set_regex_extract_all:cnn
\seq_gset_regex_extract_all:Nnn
\seq_gset_regex_extract_all:cnn
\seq_set_regex_extract_once:Nnn
\seq_set_regex_extract_once:cnn
\seq_gset_regex_extract_once:Nnn
\seq_gset_regex_extract_once:cnn
\seq_set_regex_extract_all:Nnn
\seq_set_regex_extract_all:cnn
\seq_gset_regex_extract_all:Nnn
\seq_gset_regex_extract_all:cnn
\seq_set_regex_split:Nnn
\seq_set_regex_split:cnn
\seq_gset_regex_split:Nnn
\seq_gset_regex_split:cnn
\seq_set_regex_split:NNn
\seq_set_regex_split:cNn
\seq_gset_regex_split:NNn
\seq_gset_regex_split:cNn
\seq_gset_regex_split:cNn
17496 \cs_new_protected:Npn \seq_set_regex_extract_once:Nnn #1#2#3
17497 { \regex_extract_once:nnN {#2} {#3} #1 }
17498 \cs_generate_variant:Nn \seq_set_regex_extract_once:Nnn { c }
17499 \cs_new_protected:Npn \seq_set_regex_extract_once:Nnn #1#2#3
17500 { \regex_extract_once:NnN #2 {#3} #1 }
17501 \cs_generate_variant:Nn \seq_set_regex_extract_once:NNn { c }
17502 \cs_new_protected:Npn \seq_set_regex_extract_all:Nnn #1#2#3
17503 { \regex_extract_all:nnN {#2} {#3} #1 }
17504 \cs_generate_variant:Nn \seq_set_regex_extract_all:Nnn { c }
17505 \cs_new_protected:Npn \seq_set_regex_extract_all:Nnn #1#2#3
17506 { \regex_extract_all:NnN #2 {#3} #1 }
17507 \cs_generate_variant:Nn \seq_set_regex_extract_all:NNn { c }
17508 \cs_new_protected:Npn \seq_set_regex_split:Nnn #1#2#3
17509 { \regex_split:nnN {#2} {#3} #1 }
17510 \cs_generate_variant:Nn \seq_set_regex_split:Nnn { c }
17511 \cs_new_protected:Npn \seq_set_regex_split:NNn #1#2#3
17512 { \regex_split:NnN #2 {#3} #1 }
17513 \cs_generate_variant:Nn \seq_set_regex_split:NNn { c }
17514 \group_begin:
17515   \cs_set_protected:Npn \__seq_tmp:w #1#2#3
17516   {
17517     \cs_new_protected:cpe { seq_gset_regex_ #1 :N #2 n } ##1##2##3
17518     {
17519       \group_begin:
17520         \seq_set_eq:NN \exp_not:N \l__seq_tmp_seq ##1
17521         \exp_not:c { regex_ #1 :Nn #2 }

```

```

17522         #3 {##2} {##3} \exp_not:N \l__seq_tmp_seq
17523         \seq_gset_eq:NN ##1 \exp_not:N \l__seq_tmp_seq
17524     \group_end:
17525 }
17526 \cs_generate_variant:cn { seq_gset_regex_ #1 :N #2 n } { c }
17527 }
17528 \__seq_tmp:w { extract_once } n { }
17529 \__seq_tmp:w { extract_once } N \use:n
17530 \__seq_tmp:w { extract_all } n { }
17531 \__seq_tmp:w { extract_all } N \use:n
17532 \__seq_tmp:w { split } n { }
17533 \__seq_tmp:w { split } N \use:n
17534 \group_end:

```

(End of definition for `\seq_set_regex_extract_once:Nnn` and others. These functions are documented on page 158.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

`\seq_concat:ccc`

`\seq_gconcat:NNN`

`\seq_gconcat:ccc`

```

17535 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
17536 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
17537 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
17538 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
17539 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
17540 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End of definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 159.)

`\seq_if_exist_p:N` Copies of the cs functions defined in l3basics.

`\seq_if_exist_p:c`

`\seq_if_exist:NTF`

`\seq_if_exist:cTF`

```

17541 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
17542 { TF , T , F , p }
17543 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
17544 { TF , T , F , p }

```

(End of definition for `\seq_if_exist:NTF`. This function is documented on page 159.)

63.2 Appending data to either end

`\seq_put_left:Nn` When adding to the left of a sequence, remove `\s__seq`. This is done by `\__seq_put_left_aux:w`, which also stops f-expansion.

`\seq_put_left:NV`

`\seq_put_left:Nv`

`\seq_put_left:Ne`

`\seq_put_left:No`

`\seq_put_left:Nx`

`\seq_put_left:cn`

`\seq_put_left:cV`

`\seq_put_left:cv`

`\seq_put_left:ce`

`\seq_put_left:co`

`\seq_put_left:cx`

`\seq_gput_left:Nn`

`\seq_gput_left:NV`

`\seq_gput_left:Nv`

`\seq_gput_left:Ne`

`\seq_gput_left:No`

`\seq_gput_left:Nx`

`\seq_gput_left:cn`

`\seq_gput_left:cV`

`\seq_gput_left:cv`

`\seq_gput_left:ce`

`\seq_gput_left:co`

`\seq_gput_left:cx`

```

17545 \cs_new_protected:Npn \seq_put_left:Nn #1#2
17546 {
17547     \__kernel_tl_set:Nx #1
17548     {
17549         \exp_not:n { \s__seq \__seq_item:n {#2} }
17550         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
17551     }
17552 }
17553 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
17554 {
17555     \__kernel_tl_gset:Nx #1
17556     {
17557         \exp_not:n { \s__seq \__seq_item:n {#2} }

```

```

17558         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
17559     }
17560 }
17561 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
17562 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , Ne , No , Nx }
17563 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , ce , co , cx }
17564 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , Ne , No , Nx }
17565 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , ce , co , cx }

```

(End of definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `\__seq_put_left_aux:w`. These functions are documented on page 159.)

```

\seq_put_right:Nn Since there is no trailing marker, adding an item to the right of a sequence simply means
\seq_put_right:NV wrapping it in \__seq_item:n.
\seq_put_right:Nv
\seq_put_right:Ne
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:cx
\seq_put_right:co
\seq_put_right:cx

```

```

17566 \cs_new_protected:Npn \seq_put_right:Nn #1#2
17567 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
17568 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
17569 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
17570 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , Ne , No , Nx }
17571 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , ce , co , cx }
17572 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , Ne , No , Nx }
17573 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , ce , co , cx }

```

(End of definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 159.)

```

\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:Ne
\__seq_wrap_item:n
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:cx
\seq_gput_right:co
\seq_gput_right:cx

```

63.3 Modifying sequences

This function converts its argument to a proper sequence item in an e- or x-expansion context.

```

17574 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }

```

(End of definition for `\__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```

17575 \seq_new:N \l__seq_tmp_seq

```

(End of definition for `\l__seq_tmp_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```

\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
\__seq_remove_duplicates:NN
17576 \cs_new_protected:Npn \seq_remove_duplicates:N
17577 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
17578 \cs_new_protected:Npn \seq_gremove_duplicates:N
17579 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
17580 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
17581 {
17582     \seq_clear:N \l__seq_tmp_seq
17583     \seq_map_inline:Nn #2
17584     {
17585         \seq_if_in:NnF \l__seq_tmp_seq {##1}
17586         { \seq_put_right:Nn \l__seq_tmp_seq {##1} }
17587     }
17588     #1 #2 \l__seq_tmp_seq
17589 }

```

```

17590 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
17591 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }

```

(End of definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `\__seq_remove_duplicates:NN`. These functions are documented on page 162.)

```

\seq_remove_all:Nn
\seq_remove_all:NV
\seq_remove_all:Ne
\seq_remove_all:Nx
\seq_remove_all:cn
\seq_remove_all:cV
\seq_remove_all:ce
\seq_remove_all:cx
\seq_gremove_all:Nn
\seq_gremove_all:NV
\seq_gremove_all:Ne
\seq_gremove_all:Nx
\seq_gremove_all:cn
\seq_gremove_all:cV
\seq_gremove_all:ce
\seq_gremove_all:Nx
\__seq_remove_all_aux:NNn

```

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\__seq_pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The x-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing an intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) ensures that nothing is lost.

```

17592 \cs_new_protected:Npn \seq_remove_all:Nn
17593 { \__seq_remove_all_aux:NNn \__kernel_tl_set:Nx }
17594 \cs_new_protected:Npn \seq_gremove_all:Nn
17595 { \__seq_remove_all_aux:NNn \__kernel_tl_gset:Nx }
17596 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
17597 {
17598   \__seq_push_item_def:n
17599   {
17600     \str_if_eq:nnT {##1} {#3}
17601     {
17602       \if_false: { \fi: }
17603       \tl_set:Nn \l__seq_tmpb_tl {##1}
17604       #1 #2
17605       { \if_false: } \fi:
17606       \exp_not:o {#2}
17607       \tl_if_eq:NNT \l__seq_tmpa_tl \l__seq_tmpb_tl
17608       { \use_none:nn }
17609     }
17610     \__seq_wrap_item:n {##1}
17611   }
17612   \tl_set:Nn \l__seq_tmpa_tl {#3}
17613   #1 #2 {#2}
17614   \__seq_pop_item_def:
17615 }
17616 \cs_generate_variant:Nn \seq_remove_all:Nn { NV , Ne , c , cV , ce }
17617 \cs_generate_variant:Nn \seq_remove_all:Nn { Nx , cx }
17618 \cs_generate_variant:Nn \seq_gremove_all:Nn { NV , Ne , c , cV , ce }
17619 \cs_generate_variant:Nn \seq_gremove_all:Nn { Nx , cx }

```

(End of definition for `\seq_remove_all:Nn`, `\seq_gremove_all:Nn`, and `\__seq_remove_all_aux:NNn`. These functions are documented on page 162.)

```

\__seq_int_eval:w

```

Useful to more quickly go through items.

```

17620 \cs_new_eq:NN \__seq_int_eval:w \tex_numexpr:D

```

(End of definition for `\__seq_int_eval:w`.)

```

\seq_set_item:Nnn
\seq_set_item:cnn
\seq_set_item:NnnTF
\seq_set_item:cnnTF
\seq_gset_item:Nnn
\seq_gset_item:cnn
\seq_gset_item:NnnTF
\seq_gset_item:cnnTF
\__seq_set_item:NnnNN
\__seq_set_item:nnNNNN
\__seq_set_item_false:nnNNNN
\__seq_set_item:nNnnNNNN

```

The conditionals are distinguished from the `Nnn` versions by the last argument `\use_ii:nn` vs `\use_i:nn`.


```

17621 \cs_new_protected:Npn \seq_set_item:Nnn #1#2#3
17622 { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_i:nn }
17623 \cs_new_protected:Npn \seq_gset_item:Nnn #1#2#3
17624 { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_i:nn }
17625 \cs_generate_variant:Nn \seq_set_item:Nnn { c }
17626 \cs_generate_variant:Nn \seq_gset_item:Nnn { c }
17627 \prg_new_protected_conditional:Npnn \seq_set_item:Nnn #1#2#3 { TF , T , F }
17628 { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_ii:nn }
17629 \prg_new_protected_conditional:Npnn \seq_gset_item:Nnn #1#2#3 { TF , T , F }
17630 { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_ii:nn }
17631 \prg_generate_conditional_variant:Nnn \seq_set_item:Nnn { c } { TF , T , F }
17632 \prg_generate_conditional_variant:Nnn \seq_gset_item:Nnn { c } { TF , T , F }

```

Save the item to be stored and evaluate the position and the sequence length only once. Then depending on the sign of the position, check that it is not bigger than the length (in absolute value) nor zero.

```

17633 \cs_new_protected:Npn \__seq_set_item:NnnNN #1#2#3
17634 {
17635   \tl_set:Nn \l__seq_tmpa_tl { \__seq_item:n {#3} }
17636   \exp_args:Nff \__seq_set_item:nnNNNN
17637   { \int_eval:n {#2} } { \seq_count:N #1 } #1 \use_none:nn
17638 }
17639 \cs_new_protected:Npn \__seq_set_item:nnNNNN #1#2
17640 {
17641   \int_compare:nNnTF {#1} > 0
17642   { \int_compare:nNnF {#1} > {#2} { \__seq_set_item:nNnnNNNN { #1 - 1 } } }
17643   {
17644     \int_compare:nNnF {#1} < {-#2}
17645     {
17646       \int_compare:nNnF {#1} = 0
17647       { \__seq_set_item:nNnnNNNN { #2 + #1 } }
17648     }
17649   }
17650   \__seq_set_item_false:nnNNNN {#1} {#2}
17651 }

```

If the position is not ok, `\__seq_set_item_false:nnNNNN` calls an error or returns `false` (depending on the `\use_i:nn` vs `\use_ii:nn` argument mentioned above).

```

17652 \cs_new_protected:Npn \__seq_set_item_false:nnNNNN #1#2#3#4#5#6
17653 {
17654   #6
17655   {
17656     \msg_error:nneee { seq } { item-too-large }
17657     { \token_to_str:N #3 } {#2} {#1}
17658   }
17659   { \prg_return_false: }
17660 }

```

If the position is ok, `\__seq_set_item:nNnnNNNN` makes the assignment and returns `true` (in the case of conditionals). Here `#1` is an integer expression (position minus one), it needs to be evaluated. The sequence `#5` starts with `\s__seq` (even if empty), which stops the integer expression and is absorbed by it. The `\if_meaning:w` test is slightly faster than an integer test (but only works when testing against zero, hence the offset we chose in the position). When we are done skipping items, insert the saved item `\l__seq_tmpa_tl`. For put functions the last argument of `\__seq_set_item_end:w` is

`\use_none:nn` and it absorbs the item #2 that we are removing: this is only useful for the `pop` functions.

```

17661 \cs_new_protected:Npn \__seq_set_item:nNnnNNNN #1#2#3#4#5#6#7#8
17662 {
17663   #7 #5
17664   {
17665     \s__seq
17666     \exp_after:wN \__seq_set_item:wn
17667     \int_value:w \__seq_int_eval:w #1
17668     #5 \s__seq_stop #6
17669   }
17670   #8 { } { \prg_return_true: }
17671 }
17672 \cs_new:Npn \__seq_set_item:wn #1 \__seq_item:n #2
17673 {
17674   \if_meaning:w 0 #1 \__seq_set_item_end:w \fi:
17675   \exp_not:n { \__seq_item:n {#2} }
17676   \exp_after:wN \__seq_set_item:wn
17677   \int_value:w \__seq_int_eval:w #1 - 1 \s__seq
17678 }
17679 \cs_new:Npn \__seq_set_item_end:w #1 \exp_not:n #2 #3 \s__seq #4 \s__seq_stop #5
17680 {
17681   #1
17682   \exp_not:o \l__seq_tmpa_tl
17683   \exp_not:n {#4}
17684   #5 #2
17685 }

```

(End of definition for `\seq_set_item:NnnTF` and others. These functions are documented on page 162.)

<pre> \seq_reverse:N \seq_reverse:c \seq_greverse:N \seq_greverse:c __seq_reverse:NN __seq_reverse_item:nwn </pre>	<pre> Previously, \seq_reverse:N was coded by collecting the items in reverse order after an \exp_stop_f: marker. \cs_new_protected:Npn \seq_reverse:N #1 { \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw \tl_set:Nf #2 { #2 \exp_stop_f: } } \cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f: { #2 \exp_stop_f: \@@_item:n {#1} } </pre>

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, $\text{\TeX}$'s usual tail recursion does not take place in this case: since the following `\__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, $\text{\TeX}$ cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `\__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. $\text{\TeX}$ can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

17686 \cs_new_protected:Npn \seq_reverse:N
17687 { \__seq_reverse:NN \__kernel_tl_set:Nx }
17688 \cs_new_protected:Npn \seq_greverse:N
17689 { \__seq_reverse:NN \__kernel_tl_gset:Nx }
17690 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
17691 {
17692   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
17693   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
17694   #1 #2 { #2 \exp_not:n { } }
17695   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
17696 }
17697 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
17698 {
17699   #2
17700   \exp_not:n { \__seq_item:n {#1} #3 }
17701 }
17702 \cs_generate_variant:Nn \seq_reverse:N { c }
17703 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End of definition for `\seq_reverse:N` and others. These functions are documented on page 163.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

(End of definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 163.)

`\seq_gsort:Nn`

`\seq_gsort:cn`

63.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

\seq_if_empty_p:c 17704 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
\seq_if_empty:NTF 17705 {
\seq_if_empty:cTF 17706   \if_meaning:w #1 \c_empty_seq
17707   \prg_return_true:
17708   \else:
17709   \prg_return_false:
17710   \fi:
17711 }
17712 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
17713 { c } { p , T , F , TF }

```

(End of definition for `\seq_if_empty:N`. This function is documented on page 163.)

`\seq_shuffle:N` We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive

`\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument

`\seq_gshuffle:N` divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements

`\seq_gshuffle:c` there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question

`\__seq_shuffle:NN` of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order

`\__seq_shuffle_item:n`

`\g__seq_tmp_seq`

```

17714 \seq_new:N \g__seq_tmp_seq
17715 \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
17716 \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
17717 \cs_new_protected:Npn \__seq_shuffle:NN #1#2

```

```

17718 {
17719   \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
17720   {
17721     \msg_error:nne { seq } { shuffle-too-large }
17722     { \token_to_str:N #2 }
17723   }
17724   {
17725     \group_begin:
17726     \int_zero:N \l__seq_tmpa_int
17727     \__seq_push_item_def:
17728     \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
17729     #2
17730     \__seq_pop_item_def:
17731     \seq_gclear:N \g__seq_tmp_seq
17732     \int_step_inline:nn \l__seq_tmpa_int
17733     {
17734       \seq_gput_right:Ne \g__seq_tmp_seq
17735       { \tex_the:D \tex_toks:D ##1 }
17736     }
17737     \group_end:
17738     #1 #2 \g__seq_tmp_seq
17739     \seq_gclear:N \g__seq_tmp_seq
17740   }
17741 }
17742 \cs_new_protected:Npn \__seq_shuffle_item:n
17743 {
17744   \int_incr:N \l__seq_tmpa_int
17745   \int_set:Nn \l__seq_tmpb_int
17746   { 1 + \tex_uniformdeviate:D \l__seq_tmpa_int }
17747   \tex_toks:D \l__seq_tmpa_int
17748   = \tex_toks:D \l__seq_tmpb_int
17749   \tex_toks:D \l__seq_tmpb_int
17750 }
17751 \cs_generate_variant:Nn \seq_shuffle:N { c }
17752 \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End of definition for `\seq_shuffle:N` and others. These functions are documented on page 163.)

`\seq_if_in:NnTF` The approach here is to define `\__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `\__seq_item:n` is preserved in nested situations.

```

17753 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
17754 { T , F , TF }
17755 {
17756   \group_begin:
17757   \tl_set:Nn \l__seq_tmpa_tl {#2}
17758   \cs_set_protected:Npn \__seq_item:n ##1
17759   {
17760     \tl_set:Nn \l__seq_tmpb_tl {##1}
17761     \if_meaning:w \l__seq_tmpa_tl \l__seq_tmpb_tl
17762     \exp_after:wN \__seq_if_in:
17763     \fi:

```

```

17764     }
17765     #1
17766     \group_end:
17767     \prg_return_false:
17768     \prg_break_point:
17769   }
17770   \cs_new:Npn \__seq_if_in:
17771   { \prg_break:n { \group_end: \prg_return_true: } }
17772   \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
17773   { NV , Nv , Ne , No , Nx , c , cV , cv , ce , co , cx } { T , F , TF }

```

(End of definition for \seq_if_in:NnTF and __seq_if_in:. This function is documented on page 163.)

63.5 Recovering data from sequences

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

\__seq_pop:NNNN
\__seq_pop_TF:NNNN
17774 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
17775 {
17776   \if_meaning:w #3 \c_empty_seq
17777   \tl_set:Nn #4 { \q_no_value }
17778   \else:
17779     #1#2#3#4
17780   \fi:
17781 }
17782 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
17783 {
17784   \if_meaning:w #3 \c_empty_seq
17785   % \tl_set:Nn #4 { \q_no_value }
17786   \prg_return_false:
17787   \else:
17788     #1#2#3#4
17789   \prg_return_true:
17790   \fi:
17791 }

```

(End of definition for __seq_pop:NNNN and __seq_pop_TF:NNNN.)

\seq_get_left:NN Getting an item from the left of a sequence is pretty easy: just trim off the first item after __seq_item:n at the start. We append a \q_no_value item to cover the case of an empty sequence

```

\seq_get_left:cN
\__seq_get_left:wnw
17792 \cs_new_protected:Npn \seq_get_left:NN #1#2
17793 {
17794   \__kernel_tl_set:Nx #2
17795   {
17796     \exp_after:wN \__seq_get_left:wnw
17797     #1 \__seq_item:n { \q_no_value } \s__seq_stop
17798   }
17799 }
17800 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \s__seq_stop
17801 { \exp_not:n {#2} }
17802 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End of definition for `\seq_get_left:NN` and `\__seq_get_left:wnw`. This function is documented on page 160.)

`\seq_pop_left:NN`
`\seq_pop_left:cN`
`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
`\__seq_pop_left:NNN`
`\__seq_pop_left:wnwNNN`

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```
17803 \cs_new_protected:Npn \seq_pop_left:NN
17804   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
17805 \cs_new_protected:Npn \seq_gpop_left:NN
17806   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
17807 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
17808   { \exp_after:wN \__seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
17809 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
17810   #1 \__seq_item:n #2#3 \s__seq_stop #4#5#6
17811   {
17812     #4 #5 { #1 #3 }
17813     \tl_set:Nn #6 {#2}
17814   }
17815 \cs_generate_variant:Nn \seq_pop_left:NN { c }
17816 \cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(End of definition for `\seq_pop_left:NN` and others. These functions are documented on page 160.)

`\seq_get_right:NN`
`\seq_get_right:cN`
`\__seq_get_right_loop:nw`
`\__seq_get_right_end:NnN`

First remove `\s__seq` and prepend `\q_no_value`. The first argument of `\__seq_get_right_loop:nw` is the last item found, and the second argument is empty until the end of the loop, where it is code that applies `\exp_not:n` to the last item and ends the loop.

```
17817 \cs_new_protected:Npn \seq_get_right:NN #1#2
17818   {
17819     \__kernel_tl_set:Nx #2
17820     {
17821       \exp_after:wN \use_i_ii:nnn
17822       \exp_after:wN \__seq_get_right_loop:nw
17823       \exp_after:wN \q_no_value
17824       #1
17825       \__seq_get_right_end:NnN \__seq_item:n
17826     }
17827   }
17828 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
17829   {
17830     #2 \use_none:n {#1}
17831     \__seq_get_right_loop:nw
17832   }
17833 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
17834 \cs_generate_variant:Nn \seq_get_right:NN { c }
```

(End of definition for `\seq_get_right:NN`, `\__seq_get_right_loop:nw`, and `\__seq_get_right_end:NnN`. This function is documented on page 160.)

`\seq_pop_right:NN`
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
`\__seq_pop_right:NNN`
`\__seq_pop_right_loop:nn`

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{\if_false:} \fi: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `\__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the

strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

17835 \cs_new_protected:Npn \seq_pop_right:NN
17836 { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx }
17837 \cs_new_protected:Npn \seq_gpop_right:NN
17838 { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx }
17839 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
17840 {
17841   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
17842   \cs_set_eq:NN \__seq_item:n \scan_stop:
17843   #1 #2
17844   { \if_false: } \fi: \s__seq
17845     \exp_after:wN \use_i:nnn
17846     \exp_after:wN \__seq_pop_right_loop:nn
17847     #2
17848     {
17849       \if_false: { \fi: }
17850       \__kernel_tl_set:Nx #3
17851     }
17852     { } \use_none:nn
17853     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
17854   }
17855   \cs_new:Npn \__seq_pop_right_loop:nn #1#2
17856   {
17857     #2 { \exp_not:n {#1} }
17858     \__seq_pop_right_loop:nn
17859   }
17860   \cs_generate_variant:Nn \seq_pop_right:NN { c }
17861   \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End of definition for `\seq_pop_right:NN` and others. These functions are documented on page 160.)

<code>\seq_get_left:NNTF</code> <code>\seq_get_left:cNTF</code> <code>\seq_get_right:NNTF</code> <code>\seq_get_right:cNTF</code>	Getting from the left or right with a check on the results. The first argument to <code>__seq_pop_TF:NNNN</code> is left unused. <pre> 17862 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF } 17863 { __seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 } 17864 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF } 17865 { __seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 } 17866 \prg_generate_conditional_variant:Nnn \seq_get_left:NN 17867 { c } { T , F , TF } 17868 \prg_generate_conditional_variant:Nnn \seq_get_right:NN 17869 { c } { T , F , TF } </pre>

(End of definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 161.)

<code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code> <code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code> <code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code> <code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code>	More or less the same for popping. <pre> 17870 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 17871 { T , F , TF } 17872 { __seq_pop_TF:NNNN __seq_pop_left:NNN \tl_set:Nn #1 #2 } 17873 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 17874 { T , F , TF } </pre>

```

17875 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
17876 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
17877 { T , F , TF }
17878 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
17879 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
17880 { T , F , TF }
17881 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
17882 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
17883 { T , F , TF }
17884 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
17885 { T , F , TF }
17886 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
17887 { T , F , TF }
17888 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
17889 { T , F , TF }

```

(End of definition for \seq_pop_left:NNTF and others. These functions are documented on page 161.)

\seq_item:Nn The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by **__seq_item:n** is **\prg_break:** instead of being empty, terminating the loop and returning nothing at all.

\seq_item:NV

\seq_item:Ne

\seq_item:cn

\seq_item:cV

\seq_item:ce

__seq_item:wNn

__seq_item:nN

__seq_item:nwn

```

17890 \cs_new:Npn \seq_item:Nn #1
17891 { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
17892 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
17893 {
17894   \exp_args:Nf \__seq_item:nwn
17895   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
17896   #1
17897   \prg_break: \__seq_item:n { }
17898   \prg_break_point:
17899 }
17900 \cs_new:Npn \__seq_item:nN #1#2
17901 {
17902   \int_compare:nNnTF {#1} < 0
17903   { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
17904   {#1}
17905 }
17906 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
17907 {
17908   #2
17909   \int_compare:nNnTF {#1} = 1
17910   { \prg_break:n { \exp_not:n {#3} } }
17911   { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
17912 }
17913 \cs_generate_variant:Nn \seq_item:Nn { NV , Ne , c , cV , ce }

```

(End of definition for \seq_item:Nn and others. This function is documented on page 160.)

\seq_rand_item:N Importantly, **\seq_item:Nn** only evaluates its argument once.

\seq_rand_item:c

```

17914 \cs_new:Npn \seq_rand_item:N #1
17915 {
17916   \seq_if_empty:NF #1
17917   { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }

```



```

17918     }
17919 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End of definition for `\seq_rand_item:N`. This function is documented on page 161.)

63.6 Mapping over sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

17920 \cs_new:Npn \seq_map_break:
17921 { \prg_map_break:Nn \seq_map_break: { } }
17922 \cs_new:Npn \seq_map_break:n
17923 { \prg_map_break:Nn \seq_map_break: }

```

(End of definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 165.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `\__seq_item:n`. The even-numbered arguments of `\__seq_map_function:Nw` delimited by `\__seq_item:n` are almost always empty, except at the end of the loop where it is `\prg_break:`. This allows to break the loop without needing to do a (relatively-expensive) quark test.

```

17924 \cs_new:Npn \seq_map_function:NN #1#2
17925 {
17926   \exp_after:wN \use_i_ii:nnn
17927   \exp_after:wN \__seq_map_function:Nw
17928   \exp_after:wN #2
17929   #1
17930   \prg_break:
17931   \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
17932   \prg_break_point:
17933   \prg_break_point:Nn \seq_map_break: { }
17934 }
17935 \cs_new:Npn \__seq_map_function:Nw #1
17936   #2 \__seq_item:n #3
17937   #4 \__seq_item:n #5
17938   #6 \__seq_item:n #7
17939   #8 \__seq_item:n #9
17940 {
17941   #2 #1 {#3}
17942   #4 #1 {#5}
17943   #6 #1 {#7}
17944   #8 #1 {#9}
17945   \__seq_map_function:Nw #1
17946 }
17947 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End of definition for `\seq_map_function:NN` and `\__seq_map_function:Nw`. This function is documented on page 163.)

`\__seq_push_item_def:n` The definition of `\__seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

17948 \cs_new_protected:Npn \__seq_push_item_def:n
17949 {
17950   \__seq_push_item_def:
17951   \cs_gset:Npn \__seq_item:n ##1
17952 }
17953 \cs_new_protected:Npn \__seq_push_item_def:e
17954 {
17955   \__seq_push_item_def:
17956   \cs_gset:Npe \__seq_item:n ##1
17957 }
17958 \cs_new_protected:Npn \__seq_push_item_def:
17959 {
17960   \int_gincr:N \g__kernel_prg_map_int
17961   \cs_gset_eq:cN { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
17962   \__seq_item:n
17963 }
17964 \cs_new_protected:Npn \__seq_pop_item_def:
17965 {
17966   \cs_gset_eq:Nc \__seq_item:n
17967   { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
17968   \int_gdecr:N \g__kernel_prg_map_int
17969 }

```

(End of definition for __seq_push_item_def:n, __seq_push_item_def:, and __seq_pop_item_def:.)

\seq_map_inline:Nn The idea here is that __seq_item:n is already “applied” to each item in a sequence,
\seq_map_inline:cn and so an in-line mapping is just a case of redefining __seq_item:n.

```

17970 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
17971 {
17972   \__seq_push_item_def:n {#2}
17973   #1
17974   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
17975 }
17976 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End of definition for \seq_map_inline:Nn. This function is documented on page 163.)

\seq_map_tokens:Nn This is based on the function mapping but using the same tricks as described for \prop_
\seq_map_tokens:cn map_tokens:Nn. The idea is to remove the leading \s__seq and apply the tokens such
__seq_map_tokens:nw that they are safe with the break points, hence the \use:n.

```

17977 \cs_new:Npn \seq_map_tokens:Nn #1#2
17978 {
17979   \exp_last_unbraced:Nno
17980   \use_i:nn { \__seq_map_tokens:nw {#2} } #1
17981   \prg_break:
17982   \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
17983   \prg_break_point:
17984   \prg_break_point:Nn \seq_map_break: { }
17985 }
17986 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
17987 \cs_new:Npn \__seq_map_tokens:nw #1
17988   #2 \__seq_item:n #3
17989   #4 \__seq_item:n #5
17990   #6 \__seq_item:n #7

```

```

17991     #8 \__seq_item:n #9
17992   {
17993     #2 \use:n {#1} {#3}
17994     #4 \use:n {#1} {#5}
17995     #6 \use:n {#1} {#7}
17996     #8 \use:n {#1} {#9}
17997     \__seq_map_tokens:nw {#1}
17998   }

```

(End of definition for \seq_map_tokens:Nn and __seq_map_tokens:nw. This function is documented on page 164.)

\seq_map_variable:NNn This is just a specialized version of the in-line mapping function, using an e-type expansion for the code set up so that the number of # tokens required is as expected.

\seq_map_variable:Ncn

\seq_map_variable:cNn

\seq_map_variable:ccn

```

17999 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
18000 {
18001   \__seq_push_item_def:e
18002   {
18003     \tl_set:Nn \exp_not:N #2 {##1}
18004     \exp_not:n {#3}
18005   }
18006   #1
18007   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
18008 }
18009 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
18010 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End of definition for \seq_map_variable:NNn. This function is documented on page 164.)

__seq_sep:

```

18011 \cs_new_eq:NN \__seq_sep: \__kernel_int_sep:

```

(End of definition for __seq_sep:.)

\seq_map_indexed_function:NN Similar to \seq_map_function:NN but we keep track of the item index as a ;-delimited argument of __seq_map_indexed:Nw.

\seq_map_indexed_inline:Nn

```

\__seq_map_indexed:nNn 18012 \cs_new:Npn \seq_map_indexed_function:NN #1#2
\__seq_map_indexed:Nw 18013 {
18014   \__seq_map_indexed:NN #1#2
18015   \prg_break_point:Nn \seq_map_break: { }
18016 }
18017 \cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
18018 {
18019   \int_gincr:N \g__kernel_prg_map_int
18020   \cs_gset_protected:cpn
18021   { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w } ##1##2 {#2}
18022   \exp_args:Nnc \__seq_map_indexed:NN #1
18023   { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
18024   \prg_break_point:Nn \seq_map_break:
18025   { \int_gdecr:N \g__kernel_prg_map_int }
18026 }
18027 \cs_new:Npn \__seq_map_indexed:NN #1#2
18028 {
18029   \exp_after:wN \__seq_map_indexed:Nw
18030   \exp_after:wN #2

```

```

18031     \int_value:w 1
18032     \exp_after:wN \use_i:nn
18033     \exp_after:wN \__seq_sep:
18034     #1
18035     \prg_break: \__seq_item:n { } \prg_break_point:
18036   }
18037 \cs_new:Npn \__seq_map_indexed:Nw #1#2 \__seq_sep: #3 \__seq_item:n #4
18038 {
18039   #3
18040   #1 {#2} {#4}
18041   \exp_after:wN \__seq_map_indexed:Nw
18042   \exp_after:wN #1
18043   \int_value:w \int_eval:w 1 + #2 \__seq_sep:
18044 }

```

(End of definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 164.)

`\seq_map_pairwise_function:NNN`
`\seq_map_pairwise_function:NcN`
`\seq_map_pairwise_function:cN`
`\seq_map_pairwise_function:ccN`
`\__seq_map_pairwise_function:wNN`
`\__seq_map_pairwise_function:wNw`
`\__seq_map_pairwise_function:Nnnwnn`

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s__seq \__seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

18045 \cs_new:Npn \seq_map_pairwise_function:NNN #1#2#3
18046 { \exp_after:wN \__seq_map_pairwise_function:wNN #2 \s__seq_stop #1 #3 }
18047 \cs_new:Npn \__seq_map_pairwise_function:wNN \s__seq #1 \s__seq_stop #2#3
18048 {
18049   \exp_after:wN \__seq_map_pairwise_function:wNw #2 \s__seq_stop #3
18050   #1 { ? \prg_break: } { }
18051   \prg_break_point:
18052   \prg_break_point:Nn \seq_map_break: { }
18053 }
18054 \cs_new:Npn \__seq_map_pairwise_function:wNw \s__seq #1 \s__seq_stop #2
18055 {
18056   \__seq_map_pairwise_function:Nnnwnn #2
18057   #1 { ? \prg_break: } { }
18058   \s__seq_stop
18059 }
18060 \cs_new:Npn \__seq_map_pairwise_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
18061 {
18062   \use_none:n #2
18063   \use_none:n #5
18064   #1 {#3} {#6}
18065   \__seq_map_pairwise_function:Nnnwnn #1 #4 \s__seq_stop
18066 }
18067 \cs_generate_variant:Nn \seq_map_pairwise_function:NNN { Nc , c , cc }

```

(End of definition for `\seq_map_pairwise_function:NNN` and others. This function is documented on page 164.)

`\seq_set_map_e:NNn`
`\seq_gset_map_e:NNn`
`\__seq_set_map_e:NNNn`

Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```

18068 \cs_new_protected:Npn \seq_set_map_e:NNn
18069 { \__seq_set_map_e:NNNn \__kernel_tl_set:Nx }
18070 \cs_new_protected:Npn \seq_gset_map_e:NNn
18071 { \__seq_set_map_e:NNNn \__kernel_tl_gset:Nx }
18072 \cs_new_protected:Npn \__seq_set_map_e:NNNn #1#2#3#4
18073 {
18074   \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
18075   #1 #2 { #3 }
18076   \__seq_pop_item_def:
18077 }

```

(End of definition for `\seq_set_map_e:NNn`, `\seq_gset_map_e:NNn`, and `\__seq_set_map_e:NNNn`. These functions are documented on page 166.)

`\seq_set_map:NNn` Similar to `\seq_set_map_e:NNn`, but prevents expansion of the `<inline function>`.
`\seq_gset_map:NNn`
`\__seq_set_map:NNNn`

```

18078 \cs_new_protected:Npn \seq_set_map:NNn
18079 { \__seq_set_map:NNNn \__kernel_tl_set:Nx }
18080 \cs_new_protected:Npn \seq_gset_map:NNn
18081 { \__seq_set_map:NNNn \__kernel_tl_gset:Nx }
18082 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
18083 {
18084   \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
18085   #1 #2 { #3 }
18086   \__seq_pop_item_def:
18087 }

```

(End of definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `\__seq_set_map:NNNn`. These functions are documented on page 165.)

`\seq_count:N` Since counting the items in a sequence is quite common, we optimize it by grabbing
`\seq_count:c` 8 items at a time and correspondingly adding 8 to an integer expression. At the end of
`\__seq_count:w` the loop, #9 is `\__seq_count_end:w` instead of being empty. It removes 8+ and instead
`\__seq_count_end:w` places the number of `\__seq_item:n` that `\__seq_count:w` grabbed before reaching the
end of the sequence.

```

18088 \cs_new:Npn \seq_count:N #1
18089 {
18090   \int_eval:n
18091   {
18092     \exp_after:wN \use_i:nn
18093     \exp_after:wN \__seq_count:w
18094     #1
18095     \__seq_count_end:w \__seq_item:n 7
18096     \__seq_count_end:w \__seq_item:n 6
18097     \__seq_count_end:w \__seq_item:n 5
18098     \__seq_count_end:w \__seq_item:n 4
18099     \__seq_count_end:w \__seq_item:n 3
18100     \__seq_count_end:w \__seq_item:n 2
18101     \__seq_count_end:w \__seq_item:n 1
18102     \__seq_count_end:w \__seq_item:n 0
18103     \prg_break_point:
18104   }
18105 }
18106 \cs_new:Npn \__seq_count:w
18107 #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n

```

```

18108     #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
18109     { #9 8 + \__seq_count:w }
18110 \cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
18111 \cs_generate_variant:Nn \seq_count:N { c }

```

(End of definition for `\seq_count:N`, `\__seq_count:w`, and `\__seq_count_end:w`. This function is documented on page 166.)

63.7 Using sequences

```

\seq_use:Nnnn See \clist_use:Nnnn for a general explanation. The main difference is that we use
\seq_use:cnnn \__seq_item:n as a delimiter rather than commas. We also need to add \__seq_item:n
\__seq_use:NNnNnn at various places, and \s__seq.
\__seq_use_setup:w
\__seq_use:nwwwnwn
\__seq_use:nwwn
\seq_use:Nn
\seq_use:cn
18112 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
18113 {
18114     \seq_if_exist:NTF #1
18115     {
18116         \int_case:nnF { \seq_count:N #1 }
18117         {
18118             { 0 } { }
18119             { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
18120             { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
18121         }
18122         {
18123             \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
18124             \s__seq_mark { \__seq_use:nwwwnwn {#3} }
18125             \s__seq_mark { \__seq_use:nwwn {#4} }
18126             \s__seq_stop { }
18127         }
18128     }
18129     {
18130         \msg_expandable_error:nnn
18131         { kernel } { bad-variable } {#1}
18132     }
18133 }
18134 \cs_generate_variant:Nn \seq_use:Nnnn { c }
18135 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
18136 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
18137 \cs_new:Npn \__seq_use:nwwwnwn
18138     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
18139     \s__seq_mark #6#7 \s__seq_stop #8
18140     {
18141         #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
18142         \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
18143     }
18144 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
18145     { \exp_not:n { #4 #1 #2 } }
18146 \cs_new:Npn \seq_use:Nn #1#2
18147     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
18148 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End of definition for `\seq_use:Nnnn` and others. These functions are documented on page 166.)

63.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

```

\seq_push:Nn      Pushing to a sequence is the same as adding on the left.
\seq_push:NV      18149 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\seq_push:Nv      18150 \cs_generate_variant:Nn \seq_push:Nn { NV , Nv , Ne , c , cV , cv , ce }
\seq_push:Ne      18151 \cs_generate_variant:Nn \seq_push:Nn { No , Nx , co , cx }
\seq_push:No      18152 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\seq_push:Nx      18153 \cs_generate_variant:Nn \seq_gpush:Nn { NV , Nv , Ne , c , cV , cv , ce }
\seq_push:cn      18154 \cs_generate_variant:Nn \seq_gpush:Nn { No , Nx , co , cx }
\seq_push:cV      (End of definition for \seq_push:Nn and \seq_gpush:Nn. These functions are documented on page 168.)
\seq_push:cv
\seqppsh:Nn      In most cases, getting items from the stack does not need to specify that this is from the
\seqppsh:cn      left. So alias are provided.
\seqppsh:Nn      18155 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seqppsh:cn      18156 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seqgppsh:Nn      18157 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seqgppsh:cn      18158 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\seq_gpush:Ne      18159 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\seq_gpush:No      18160 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
\seq_gpush:Nx      (End of definition for \seq_get:NN, \seq_pop:NN, and \seq_gpop:NN. These functions are documented
\seq_gpush:cn      on page 167.)
\seq_gpush:cV
\seq_get:NNTF      More copies.
\seq_gpush:cv      18161 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_get:NNTF      18162 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:NNTF      18163 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF      18164 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
\seq_gpop:cNTF      18165 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
\seq_gpop:cNTF      18166 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

(End of definition for \seq_get:NNTF, \seq_pop:NNTF, and \seq_gpop:NNTF. These functions are docu-
mented on page 167.)

```

63.9 Viewing sequences

```

\seq_show:N      Apply the general \__kernel_chk_tl_type:NnnT.
\seq_show:c      18167 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nneeee }
\seq_log:N       18168 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c       18169 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nneeee }
\__seq_show:NN   18170 \cs_generate_variant:Nn \seq_log:N { c }
\__seq_show_validate:nn 18171 \cs_new_protected:Npn \__seq_show:NN #1#2
                  18172 {
                  18173   \__kernel_chk_tl_type:NnnT #2 { seq }
                  18174   {
                  18175     \s__seq
                  18176     \exp_after:wN \use_i:nn \exp_after:wN \__seq_show_validate:nn #2
                  18177     \q_recursion_tail \q_recursion_tail \q_recursion_stop
                  18178   }
                  18179   {
                  18180     #1 { seq } { show }

```

```

18181         { \token_to_str:N #2 }
18182         { \seq_map_function:NN #2 \msg_show_item:n }
18183         { } { }
18184     }
18185 }
18186 \cs_new:Npn \__seq_show_validate:nn #1#2
18187 {
18188     \quark_if_recursion_tail_stop:n {#2}
18189     \__seq_wrap_item:n {#2}
18190     \__seq_show_validate:nn
18191 }

```

(End of definition for `\seq_show:N` and others. These functions are documented on page 170.)

63.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.
`\l_tmpb_seq` `\seq_new:N \l_tmpa_seq`
`\g_tmpa_seq` `\seq_new:N \l_tmpb_seq`
`\g_tmpb_seq` `\seq_new:N \g_tmpa_seq`
`\seq_new:N \g_tmpb_seq`

(End of definition for `\l_tmpa_seq` and others. These variables are documented on page 170.)

```

18196 </code>

```


Chapter 64

l3int implementation

18197 `\*code)`

18198 `<@@=int>`

The following test files are used for this code: m3int001,m3int002,m3int03.

`\c_max_register_int` Done in l3basics.

(End of definition for \c_max_register_int. This variable is documented on page 184.)

`\__int_to_roman:w` Done in l3basics.

`\if_int_compare:w` *(End of definition for __int_to_roman:w and \if_int_compare:w. This function is documented on page 185.)*

`\or:` Done in l3basics.

(End of definition for \or:. This function is documented on page 185.)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

<code>__int_eval:w</code>	18199 <code>\cs_new_eq:NN \int_value:w \tex_number:D</code>
<code>__int_eval_end:</code>	18200 <code>\cs_new_eq:NN __int_eval:w \tex_numexpr:D</code>
<code>\if_int_odd:w</code>	18201 <code>\cs_new_eq:NN __int_eval_end: \tex_relax:D</code>
<code>\if_case:w</code>	18202 <code>\cs_new_eq:NN \if_int_odd:w \tex_ifodd:D</code>
	18203 <code>\cs_new_eq:NN \if_case:w \tex_ifcase:D</code>

(End of definition for \int_value:w and others. These functions are documented on page 185.)

`\s__int_mark` Scan marks used throughout the module.

<code>\s__int_stop</code>	18204 <code>\scan_new:N \s__int_mark</code>
	18205 <code>\scan_new:N \s__int_stop</code>

(End of definition for \s__int_mark and \s__int_stop.)

`\__int_use_none_delimit_by_s_stop:w` Function to gobble until a scan mark.

18206 `\cs_new:Npn \__int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }`

(End of definition for __int_use_none_delimit_by_s_stop:w.)

`\q__int_recursion_tail` Quarks for recursion.

<code>\q__int_recursion_stop</code>	18207 <code>\quark_new:N \q__int_recursion_tail</code>
	18208 <code>\quark_new:N \q__int_recursion_stop</code>

(End of definition for `\q__int_recursion_tail` and `\q__int_recursion_stop`.)

`\__int_if_recursion_tail_stop_do:Nn`
`\__int_if_recursion_tail_stop:N`

Functions to query quarks.

```
18209 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop_do:Nn
18210 \__kernel_quark_new_test:N \__int_if_recursion_tail_stop:N
```

(End of definition for `\__int_if_recursion_tail_stop_do:Nn` and `\__int_if_recursion_tail_stop:N`.)

64.1 Integer expressions

`\int_eval:n` Wrapper for `\__int_eval:w`: can be used in an integer expression or directly in the input stream. It is very slightly faster to use `\the` rather than `\number` to turn the expression to a number. When debugging, we introduce parentheses to catch early termination (see `!3debug`).

```
18211 \cs_new:Npn \int_eval:n #1
18212 { \tex_the:D \__int_eval:w #1 \__int_eval_end: }
18213 \cs_new:Npn \int_eval:w { \tex_the:D \__int_eval:w }
```

(End of definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 172.)

`\__int_sep:`

```
18214 \cs_new_eq:NN \__int_sep: \__kernel_int_sep:
```

(End of definition for `\__int_sep:`.)

`\int_sign:n` See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w ... \exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

`\__int_sign:Nw`

```
18215 \cs_new:Npn \int_sign:n #1
18216 {
18217   \int_value:w \exp_after:wN \__int_sign:Nw
18218   \int_value:w \__int_eval:w #1 \__int_eval_end: \__int_sep:
18219   \exp_stop_f:
18220 }
18221 \cs_new:Npn \__int_sign:Nw #1#2 \__int_sep:
18222 {
18223   \if_meaning:w 0 #1
18224   0
18225   \else:
18226     \if_meaning:w - #1 - \fi: 1
18227   \fi:
18228 }
```

(End of definition for `\int_sign:n` and `\__int_sign:Nw`. This function is documented on page 172.)

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

`\__int_abs:N`

`\int_max:nn`

`\int_min:nn`

`\__int_maxmin:wwN`

```
18229 \cs_new:Npn \int_abs:n #1
18230 {
18231   \int_value:w \exp_after:wN \__int_abs:N
18232   \int_value:w \__int_eval:w #1 \__int_eval_end:
18233   \exp_stop_f:
```

```

18234 }
18235 \cs_new:Npn \__int_abs:N #1
18236 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
18237 \cs_new:Npn \int_max:nn #1#2
18238 {
18239   \int_value:w \exp_after:wN \__int_maxmin:wwN
18240   \int_value:w \__int_eval:w #1 \exp_after:wN \__int_sep:
18241   \int_value:w \__int_eval:w #2 \__int_sep:
18242   >
18243   \exp_stop_f:
18244 }
18245 \cs_new:Npn \int_min:nn #1#2
18246 {
18247   \int_value:w \exp_after:wN \__int_maxmin:wwN
18248   \int_value:w \__int_eval:w #1 \exp_after:wN \__int_sep:
18249   \int_value:w \__int_eval:w #2 \__int_sep:
18250   <
18251   \exp_stop_f:
18252 }
18253 \cs_new:Npn \__int_maxmin:wwN #1 \__int_sep: #2 \__int_sep: #3
18254 {
18255   \if_int_compare:w #1 #3 #2 ~
18256   #1
18257   \else:
18258   #2
18259   \fi:
18260 }

```

(End of definition for `\int_abs:n` and others. These functions are documented on page 172.)

`\int_div_truncate:nn` As `\__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(| \#3\#4 | - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behavior that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

18261 \cs_new:Npn \int_div_truncate:nn #1#2
18262 {
18263   \int_value:w \__int_eval:w
18264   \exp_after:wN \__int_div_truncate:NwNw
18265   \int_value:w \__int_eval:w #1 \exp_after:wN \__int_sep:
18266   \int_value:w \__int_eval:w #2 \__int_sep:
18267   \__int_eval_end:
18268 }
18269 \cs_new:Npn \__int_div_truncate:NwNw #1#2 \__int_sep: #3#4 \__int_sep:
18270 {
18271   \if_meaning:w 0 #1
18272   0
18273   \else:
18274   (
18275     #1#2

```

```

18276     \if_meaning:w - #1 + \else: - \fi:
18277     ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
18278   )
18279   \fi:
18280   / #3#4
18281 }

```

For the sake of completeness:

```

18282 \cs_new:Npn \int_div_round:nn #1#2
18283 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

18284 \cs_new:Npn \int_mod:nn #1#2
18285 {
18286   \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
18287   \int_value:w \__int_eval:w #1 \exp_after:wN \__int_sep:
18288   \int_value:w \__int_eval:w #2 \__int_sep:
18289   \__int_eval_end:
18290 }
18291 \cs_new:Npn \__int_mod:ww #1 \__int_sep: #2 \__int_sep:
18292 { #1 - ( \__int_div_truncate:NwNw #1 \__int_sep: #2 \__int_sep: ) * #2 }

```

(End of definition for `\int_div_truncate:nn` and others. These functions are documented on page 172.)

`\__kernel_int_add:nnn` Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31} + 1, 2^{31} - 1]$. The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in $[-2^{31} + 1, -1]$ and the other in $[0, 2^{31} - 1]$) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```

18293 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
18294 {
18295   \int_value:w \__int_eval:w #1
18296   \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
18297   \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
18298   \__int_eval_end:
18299 }

```

(End of definition for `\__kernel_int_add:nnn`.)

64.2 Creating and initializing integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\newcount` (and other allocators) are `\outer:` to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```

18300 \cs_new_protected:Npn \int_new:N #1
18301 {
18302   \__kernel_chk_if_free_cs:N #1
18303   \cs:w newcount \cs_end: #1
18304 }
18305 \cs_generate_variant:Nn \int_new:N { c }

```

(End of definition for `\int_new:N`. This function is documented on page 173.)

`\int_const:Nn` As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

`\int_const:cn`

`\__int_const:nN`

`\__int_const:eN`

`\__int_constdef:Nw`

`\c_int_max_constdef_int`

```

18306 \cs_new_protected:Npn \int_const:Nn #1#2
18307   { \__int_const:eN { \int_eval:n {#2} } #1 }
18308 \cs_generate_variant:Nn \int_const:Nn { c }
18309 \cs_new_protected:Npn \__int_const:nN #1#2
18310   {
18311     \int_compare:nNnTF {#1} < \c_zero_int
18312     {
18313       \int_new:N #2
18314       \tex_global:D
18315     }
18316     {
18317       \int_compare:nNnTF {#1} > \c_int_max_constdef_int
18318       {
18319         \int_new:N #2
18320         \tex_global:D
18321       }
18322       {
18323         \__kernel_chk_if_free_cs:N #2
18324         \tex_global:D \__int_constdef:Nw
18325       }
18326     }
18327     #2 = \__int_eval:w #1 \__int_eval_end:
18328   }
18329 \cs_generate_variant:Nn \__int_const:nN { e }
18330 \if_int_odd:w 0
18331   \cs_if_exist:NT \tex_luatexversion:D { 1 }
18332   \cs_if_exist:NT \tex_omathchardef:D { 1 }
18333   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
18334   \cs_if_exist:NTF \tex_omathchardef:D
18335     { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
18336     { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
18337   \tex_global:D \__int_constdef:Nw \c_int_max_constdef_int 1114111 ~
18338 \else:
18339   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
18340   \tex_global:D \__int_constdef:Nw \c_int_max_constdef_int 32767 ~
18341 \fi:

```

(End of definition for `\int_const:Nn` and others. This function is documented on page 173.)

`\int_zero:N` Functions that reset an `<integer>` register to zero.

`\int_zero:c`

`\int_gzero:N`

`\int_gzero:c`

```

18342 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }
18343 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }
18344 \cs_generate_variant:Nn \int_zero:N { c }
18345 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End of definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 173.)

\int_zero_new:N Create a register if needed, otherwise clear it.

\int_zero_new:c 18346 \cs_new_protected:Npn \int_zero_new:N #1

\int_gzero_new:N 18347 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }

\int_gzero_new:c 18348 \cs_new_protected:Npn \int_gzero_new:N #1

18349 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }

18350 \cs_generate_variant:Nn \int_zero_new:N { c }

18351 \cs_generate_variant:Nn \int_gzero_new:N { c }

(End of definition for \int_zero_new:N and \int_gzero_new:N. These functions are documented on page 173.)

\int_set_eq:NN Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as TeX does it for us.

\int_set_eq:cN

\int_set_eq:Nc

\int_set_eq:cc 18352 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }

\int_gset_eq:NN 18353 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }

\int_gset_eq:cN 18354 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }

\int_gset_eq:Nc 18355 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

\int_gset_eq:cc

(End of definition for \int_set_eq:NN and \int_gset_eq:NN. These functions are documented on page 173.)

\int_if_exist_p:N Copies of the cs functions defined in l3basics.

\int_if_exist_p:c 18356 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N

\int_if_exist:NTF 18357 { TF , T , F , p }

\int_if_exist:cTF 18358 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c

18359 { TF , T , F , p }

(End of definition for \int_if_exist:NTF. This function is documented on page 173.)

64.3 Setting and incrementing integers

\int_add:Nn Adding and subtracting to and from a counter. Including here the optional by would slow down these operations by a few percent.

\int_add:cn

\int_gadd:Nn 18360 \cs_new_protected:Npn \int_add:Nn #1#2

\int_gadd:cn 18361 { \tex_advance:D #1 __int_eval:w #2 __int_eval_end: }

\int_sub:Nn 18362 \cs_new_protected:Npn \int_sub:Nn #1#2

\int_sub:cn 18363 { \tex_advance:D #1 - __int_eval:w #2 __int_eval_end: }

\int_gsub:Nn 18364 \cs_new_protected:Npn \int_gadd:Nn #1#2

\int_gsub:cn 18365 { \tex_global:D \tex_advance:D #1 __int_eval:w #2 __int_eval_end: }

18366 \cs_new_protected:Npn \int_gsub:Nn #1#2

18367 { \tex_global:D \tex_advance:D #1 - __int_eval:w #2 __int_eval_end: }

18368 \cs_generate_variant:Nn \int_add:Nn { c }

18369 \cs_generate_variant:Nn \int_gadd:Nn { c }

18370 \cs_generate_variant:Nn \int_sub:Nn { c }

18371 \cs_generate_variant:Nn \int_gsub:Nn { c }

(End of definition for \int_add:Nn and others. These functions are documented on page 174.)

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.

\int_incr:c 18372 \cs_new_protected:Npn \int_incr:N #1

\int_gincr:N 18373 { \tex_advance:D #1 \c_one_int }

\int_gincr:c 18374 \cs_new_protected:Npn \int_decr:N #1

\int_decr:N 18375 { \tex_advance:D #1 - \c_one_int }

\int_decr:c

\int_gdecr:N

\int_gdecr:c

```

18376 \cs_new_protected:Npn \int_gincr:N #1
18377 { \tex_global:D \tex_advance:D #1 \c_one_int }
18378 \cs_new_protected:Npn \int_gdecr:N #1
18379 { \tex_global:D \tex_advance:D #1 - \c_one_int }
18380 \cs_generate_variant:Nn \int_incr:N { c }
18381 \cs_generate_variant:Nn \int_decr:N { c }
18382 \cs_generate_variant:Nn \int_gincr:N { c }
18383 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End of definition for `\int_incr:N` and others. These functions are documented on page 174.)

```

\int_set:Nn As integers are register-based TEX issues an error if they are not defined. While the =
\int_set:cn sign is optional, this version with = is slightly quicker than without, while adding the
\int_set:NV optional space after = slows things down minutely.
\int_set:cV
\int_gset:Nn 18384 \cs_new_protected:Npn \int_set:Nn #1#2
\int_gset:cn 18385 { #1 = \__int_eval:w #2 \__int_eval_end: }
\int_gset:NV 18386 \cs_new_protected:Npn \int_gset:Nn #1#2
\int_gset:cV 18387 { \tex_global:D #1 = \__int_eval:w #2 \__int_eval_end: }
18388 \cs_generate_variant:Nn \int_set:Nn { NV , c , cV }
18389 \cs_generate_variant:Nn \int_gset:Nn { NV , c , cV }

```

(End of definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 174.)

```

\int_set_regex_count:Nnn
\int_set_regex_count:cn 18390 \cs_new_protected:Npn \int_set_regex_count:Nnn #1#2#3
\int_gset_regex_count:Nnn 18391 { \regex_count:nnN {#2} {#3} #1 }
\int_gset_regex_count:cn 18392 \cs_generate_variant:Nn \int_set_regex_count:Nnn { c }
\int_set_regex_count:NNn 18393 \cs_new_protected:Npn \int_gset_regex_count:Nnn #1#2#3
\int_set_regex_count:cNn 18394 {
\int_gset_regex_count:NNn 18395 \group_begin:
18396 \int_set_eq:NN \l__int_tmpa_int #1
18397 \regex_count:nnN {#2} {#3} \l__int_tmpa_int
18398 \int_gset_eq:NN #1 \l__int_tmpa_int
18399 \group_end:
18400 }
18401 \cs_generate_variant:Nn \int_gset_regex_count:Nnn { c }
18402 \cs_new_protected:Npn \int_set_regex_count:NNn #1#2#3
18403 { \regex_count:NnN #2 {#3} #1 }
18404 \cs_generate_variant:Nn \int_set_regex_count:Nnn { c }
18405 \cs_new_protected:Npn \int_gset_regex_count:NNn #1#2#3
18406 {
18407 \group_begin:
18408 \int_set_eq:NN \l__int_tmpa_int #1
18409 \regex_count:NnN #2 {#3} \l__int_tmpa_int
18410 \int_gset_eq:NN #1 \l__int_tmpa_int
18411 \group_end:
18412 }
18413 \cs_generate_variant:Nn \int_gset_regex_count:NNn { c }

```

(End of definition for `\int_set_regex_count:Nnn` and others. These functions are documented on page 174.)

64.4 Using integers

`\int_use:N` Here is how counters are accessed. We hand-code the `c` variant for some speed gain.

```
\int_use:c
18414 \cs_new_eq:NN \int_use:N \tex_the:D
18415 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End of definition for `\int_use:N`. This function is documented on page 175.)

64.5 Integer expression conditionals

`\__int_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `\__int_compare_error:.` This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `\__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
18416 \cs_new_protected:Npn \__int_compare_error:
18417 {
18418   \if_int_compare:w \c_zero_int \c_zero_int \fi:
18419   =
18420   \__int_compare_error:
18421 }
18422 \cs_new:Npn \__int_compare_error:Nw
18423 #1#2 \s__int_stop
18424 {
18425   { }
18426   \c_zero_int \fi:
18427   \msg_expandable_error:nnn
18428   { kernel } { unknown-comparison } {#1}
18429   \prg_return_false:
18430 }
```

(End of definition for `\__int_compare_error:` and `\__int_compare_error:Nw`.)

```
\int_compare_p:n
\int_compare:nTF
\__int_compare:w
\__int_compare:Nw
\__int_compare:NNw
\__int_compare:nnN
\__int_compare_end:=:NNw
\__int_compare_=:NNw
\__int_compare_<:NNw
\__int_compare_>:NNw
\__int_compare_==:NNw
\__int_compare_!=:NNw
\__int_compare_<=:NNw
\__int_compare_>=:NNw
```

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `\__int_compare:Nw` reads one *operand* and one *comparison* symbol, and leaves roughly

```

\__int_compare_end:=:NNw      <operand> \prg_return_false: \fi:
\__int_compare_=:NNw         \reverse_if:N \if_int_compare:w <operand> <comparison>
\__int_compare_<:NNw         \__int_compare:Nw
```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *comparisons* is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let $\text{\TeX}$ evaluate this left hand side of the (in)equality using _int_eval:w . Since the relation symbols $<$, $>$, $=$ and $!$ are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, $\text{\_int\_compare\_error:}$ is expanded, inserting $=$ and itself after an error. In all cases, _int_compare:w receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items e and _nd_ , with a trailing _s_int_stop used to grab the entire argument when necessary.

```

18431 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
18432 {
18433     \exp_after:wN \_int_compare:w
18434     \int_value:w \_int_eval:w #1 \_int_compare_error:
18435 }
18436 \cs_new:Npn \_int_compare:w #1 \_int_compare_error:
18437 {
18438     \exp_after:wN \if_false: \int_value:w
18439     \_int_compare:Nw #1 e { = nd_ } \_s_int_stop
18440 }

```

The goal here is to find an $\langle operand \rangle$ and a $\langle comparison \rangle$. The $\langle operand \rangle$ is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying _int_to_roman:w , after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with _int_compare:NNw to determine the relation symbol, building a control sequence from it (_token_to_str:N gives better errors if $\#1$ is not a character). All the extended forms have an extra $=$ hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by $\text{\TeX}$ into _scan_stop: , ignored thanks to _unexpanded , and $\text{\_int\_compare\_error:Nw}$ raises an error.

```

18441 \cs_new:Npn \_int_compare:Nw #1#2 \_s_int_stop
18442 {
18443     \exp_after:wN \_int_compare:NNw
18444     \_int_to_roman:w - 0 #2 \_s_int_mark
18445     #1#2 \_s_int_stop
18446 }
18447 \cs_new:Npn \_int_compare:NNw #1#2#3 \_s_int_mark
18448 {
18449     \_kernel_exp_not:w
18450     \use:c
18451     {
18452         \_int_compare_ \token_to_str:N #1
18453         \if_meaning:w = #2 = \fi:
18454         :NNw
18455     }
18456     \_int_compare_error:Nw #1
18457 }

```

When the last $\langle operand \rangle$ is seen, _int_compare:NNw receives e and _nd_ as arguments, hence calling $\text{\_int\_compare\_end\_=:NNw}$ to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls _int_compare:nnN where $\#1$ is _if_int_compare:w or _reverse_if:N _if_int_compare:w , $\#2$ is the $\langle operand \rangle$, and $\#3$ is one of $<$, $=$,

or >. As announced earlier, we leave the *<operand>* for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return false. Otherwise, we apply the conditional #1 to the *<operand>* #2 and the comparison #3, and call `\__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

18458 \cs_new:cpn { __int_compare_end=:NNw } #1#2#3 e #4 \s__int_stop
18459 {
18460   {#3} \exp_stop_f:
18461   \prg_return_false: \else: \prg_return_true: \fi:
18462 }
18463 \cs_new:Npn \__int_compare:nnN #1#2#3
18464 {
18465   {#2} \exp_stop_f:
18466   \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
18467   \fi:
18468   #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
18469 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\__int_compare_error:Nw` (*token*) responsible for error detection.

```

18470 \cs_new:cpn { __int_compare=:NNw } #1#2#3 =
18471 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
18472 \cs_new:cpn { __int_compare:<:NNw } #1#2#3 <
18473 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
18474 \cs_new:cpn { __int_compare:>:NNw } #1#2#3 >
18475 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
18476 \cs_new:cpn { __int_compare:=:NNw } #1#2#3 ==
18477 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
18478 \cs_new:cpn { __int_compare!=:NNw } #1#2#3 !=
18479 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
18480 \cs_new:cpn { __int_compare<=:NNw } #1#2#3 <=
18481 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
18482 \cs_new:cpn { __int_compare>=:NNw } #1#2#3 >=
18483 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End of definition for `\int_compare:nTF` and others. This function is documented on page 176.)

`\int_compare_p:nNn` More efficient but less natural in typing; we add hand-tuned variants as the build-in behavior makes this much more efficient. No mappings yet so the setup has to be written out in full.

```

\int_compare_p:nNn
\int_compare_p:vNn
\int_compare_p:nNv
\int_compare_p:vNv
\int_compare:nNnTF
\int_compare:vNnTF
\int_compare:nNvTF
\int_compare:vNvTF
18484 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
18485 {
18486   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
18487   \prg_return_true:
18488   \else:
18489   \prg_return_false:
18490   \fi:
18491 }
18492 \cs_new:Npn \int_compare_p:vNn #1#2#3
18493 { \int_compare_p:nNn { \use:c {#1} } #2 {#3} }
18494 \cs_new:Npn \int_compare_p:nNv #1#2#3
18495 { \int_compare_p:nNn {#1} #2 { \use:c {#3} } }
18496 \cs_new:Npn \int_compare_p:vNv #1#2#3

```

```

18497 { \int_compare_p:nNn { \use:c {#1} } #2 { \use:c {#3} } }
18498 \cs_new:Npn \int_compare:vNnT #1#2#3
18499 { \int_compare:nNnT { \use:c {#1} } #2 {#3} }
18500 \cs_new:Npn \int_compare:nNvT #1#2#3
18501 { \int_compare:nNnT {#1} #2 { \use:c {#3} } }
18502 \cs_new:Npn \int_compare:vNvT #1#2#3
18503 { \int_compare:nNnT { \use:c {#1} } #2 { \use:c {#3} } }
18504 \cs_new:Npn \int_compare:vNnF #1#2#3
18505 { \int_compare:nNnF { \use:c {#1} } #2 {#3} }
18506 \cs_new:Npn \int_compare:nNvF #1#2#3
18507 { \int_compare:nNnF {#1} #2 { \use:c {#3} } }
18508 \cs_new:Npn \int_compare:vNvF #1#2#3
18509 { \int_compare:nNnF { \use:c {#1} } #2 { \use:c {#3} } }
18510 \cs_new:Npn \int_compare:vNnTF #1#2#3
18511 { \int_compare:nNnTF { \use:c {#1} } #2 {#3} }
18512 \cs_new:Npn \int_compare:nNvTF #1#2#3
18513 { \int_compare:nNnTF {#1} #2 { \use:c {#3} } }
18514 \cs_new:Npn \int_compare:vNvTF #1#2#3
18515 { \int_compare:nNnTF { \use:c {#1} } #2 { \use:c {#3} } }

```

(End of definition for \int_compare:nNnTF. This function is documented on page 175.)

\int_if_zero_p:n
\int_if_zero:nTF

```

18516 \prg_new_conditional:Npnn \int_if_zero:n #1 { p , T , F , TF }
18517 {
18518   \if_int_compare:w \__int_eval:w #1 = \c_zero_int
18519     \prg_return_true:
18520   \else:
18521     \prg_return_false:
18522   \fi:
18523 }

```

(End of definition for \int_if_zero:nTF. This function is documented on page 177.)

\int_case:nn
\int_case:nnTF
__int_case:nnTF
__int_case:nw
__int_case_end:nw

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for \str_case:nnTF as described in l3str.

```

18524 \cs_new:Npn \int_case:nnTF #1
18525 {
18526   \exp:w
18527   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
18528 }
18529 \cs_new:Npn \int_case:nnT #1#2#3
18530 {
18531   \exp:w
18532   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
18533 }
18534 \cs_new:Npn \int_case:nnF #1#2
18535 {
18536   \exp:w
18537   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
18538 }
18539 \cs_new:Npn \int_case:nn #1#2
18540 {
18541   \exp:w

```

```

18542     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
18543   }
18544   \cs_new:Npn \__int_case:nnTF #1#2#3#4
18545     { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
18546   \cs_new:Npn \__int_case:nw #1#2#3
18547     {
18548       \int_compare:nNnTF {#1} = {#2}
18549       { \__int_case_end:nw {#3} }
18550       { \__int_case:nw {#1} }
18551     }
18552   \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
18553     { \exp_end: #1 #4 }

```

(End of definition for `\int_case:nnTF` and others. This function is documented on page 177.)

```

\int_if_odd_p:n A predicate function.
\int_if_odd:nTF
\int_if_even_p:n
\int_if_even:nTF
18554 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
18555   {
18556     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
18557     \prg_return_true:
18558   \else:
18559     \prg_return_false:
18560   \fi:
18561   }
18562 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
18563   {
18564     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
18565     \prg_return_true:
18566   \else:
18567     \prg_return_false:
18568   \fi:
18569   }

```

(End of definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 177.)

64.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn
\int_do_while:nn
\int_do_until:nn
18570 \cs_new:Npn \int_while_do:nn #1#2
18571   {
18572     \int_compare:nT {#1}
18573     {
18574       #2
18575       \int_while_do:nn {#1} {#2}
18576     }
18577   }
18578 \cs_new:Npn \int_until_do:nn #1#2
18579   {
18580     \int_compare:nF {#1}
18581     {
18582       #2

```

```

18583         \int_until_do:nn {#1} {#2}
18584     }
18585 }
18586 \cs_new:Npn \int_do_while:nn #1#2
18587 {
18588     #2
18589     \int_compare:nT {#1}
18590     { \int_do_while:nn {#1} {#2} }
18591 }
18592 \cs_new:Npn \int_do_until:nn #1#2
18593 {
18594     #2
18595     \int_compare:nF {#1}
18596     { \int_do_until:nn {#1} {#2} }
18597 }

```

(End of definition for `\int_while_do:nn` and others. These functions are documented on page 178.)

```

\int_while_do:nNnn
\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn

```

As above but not using the more natural syntax.

```

18598 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
18599 {
18600     \int_compare:nNnT {#1} #2 {#3}
18601     {
18602         #4
18603         \int_while_do:nNnn {#1} #2 {#3} {#4}
18604     }
18605 }
18606 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
18607 {
18608     \int_compare:nNnF {#1} #2 {#3}
18609     {
18610         #4
18611         \int_until_do:nNnn {#1} #2 {#3} {#4}
18612     }
18613 }
18614 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
18615 {
18616     #4
18617     \int_compare:nNnT {#1} #2 {#3}
18618     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
18619 }
18620 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
18621 {
18622     #4
18623     \int_compare:nNnF {#1} #2 {#3}
18624     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
18625 }

```

(End of definition for `\int_while_do:nNnn` and others. These functions are documented on page 178.)

64.7 Integer step functions

```

\int_step_function:nnnN
    \__int_step:w
    \__int_step:Nw
\int_step_function:nN
\int_step_function:nnN

```

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the

start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step). And since when we're doing the test the step size is the result of `\__int_eval:w` we know that only the value 0 has a leading token 0 which we can use for a faster test than `\int_compare:nNnTF`.

```

18626 \cs_new:Npn \int_step_function:nnnN #1#2#3
18627 {
18628     \exp_after:wN \__int_step:w
18629     \int_value:w \__int_eval:w #1 \exp_after:wN \__int_sep:
18630     \int_value:w \__int_eval:w #2 \exp_after:wN \__int_sep:
18631     \int_value:w \__int_eval:w #3 \__int_sep:
18632 }
18633 \cs_new:Npn \__int_step:w #1 \__int_sep: #2 \__int_sep: #3 \__int_sep: #4
18634 {
18635     \int_compare:nNnTF {#2} > \c_zero_int
18636     { \__int_step:Nw > }
18637     {
18638         \if_meaning:w 0 #2
18639         \exp_after:wN \use_ii:nn
18640         \fi:
18641         \use_none:n
18642         {
18643             \msg_expandable_error:nnn
18644             { kernel } { zero-step } {#4}
18645             \prg_break:
18646         }
18647         \__int_step:Nw <
18648     }
18649     #1 \__int_sep: {#2} {#3} {#4}
18650     \prg_break_point:
18651 }
18652 \cs_new:Npn \__int_step:Nw #1#2 \__int_sep: #3#4#5
18653 {
18654     \if_int_compare:w #2 #1 #4 \exp_stop_f:
18655     \prg_break:n
18656     \fi:
18657     #5 {#2}
18658     \exp_after:wN \__int_step:Nw
18659     \exp_after:wN #1
18660     \int_value:w \__int_eval:w #2 + #3 \__int_sep: {#3} {#4} {#5}
18661 }
18662 \cs_new:Npn \int_step_function:nN
18663 { \int_step_function:nnnN \c_one_int \c_one_int }
18664 \cs_new:Npn \int_step_function:nnN #1
18665 { \int_step_function:nnnN {#1} \c_one_int }

```

(End of definition for `\int_step_function:nnnN` and others. These functions are documented on page 179.)

`\int_step_tokens:nn` Because the internals `\__int_step:wwn` and `\__int_step:Nwnnn` are defined in such a way that they work with both a single token or a braced group of tokens these are really the same as the function variants.

```

18666 \cs_new_eq:NN \int_step_tokens:nn \int_step_function:nN
18667 \cs_new_eq:NN \int_step_tokens:nnn \int_step_function:nnN

```

```
18668 \cs_new_eq:NN \int_step_tokens:nnnn \int_step_function:nnnN
```

(End of definition for `\int_step_tokens:nn`, `\int_step_tokens:nnn`, and `\int_step_tokens:nnnn`. These functions are documented on page 179.)

```
\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
\__int_step:NNnnnn
```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```
18669 \cs_new_protected:Npn \int_step_inline:nn
18670 { \int_step_inline:nnnn { 1 } { 1 } }
18671 \cs_new_protected:Npn \int_step_inline:nnn #1
18672 { \int_step_inline:nnnn {#1} { 1 } }
18673 \cs_new_protected:Npn \int_step_inline:nnnn
18674 {
18675   \int_gincr:N \g__kernel_prg_map_int
18676   \exp_args:NNc \__int_step:NNnnnn
18677   \cs_gset_protected:Npn
18678     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
18679 }
18680 \cs_new_protected:Npn \int_step_variable:nNn
18681 { \int_step_variable:nnnNn { 1 } { 1 } }
18682 \cs_new_protected:Npn \int_step_variable:nnNn #1
18683 { \int_step_variable:nnnNn {#1} { 1 } }
18684 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
18685 {
18686   \int_gincr:N \g__kernel_prg_map_int
18687   \exp_args:NNc \__int_step:NNnnnn
18688   \cs_gset_protected:Npe
18689     { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
18690     {#1}{#2}{#3}
18691     {
18692       \tl_set:Nn \exp_not:N #4 {##1}
18693       \exp_not:n {#5}
18694     }
18695 }
18696 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
18697 {
18698   #1 #2 ##1 {#6}
18699   \int_step_function:nnnN {#3} {#4} {#5} #2
18700   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
18701 }
```

(End of definition for `\int_step_inline:nn` and others. These functions are documented on page 179.)

64.8 Formatting integers

```
\int_to_arabic:n
\int_to_arabic:v
```

Nothing exciting here.

```
18702 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
18703 \cs_generate_variant:Nn \int_to_arabic:n { v }
```

(End of definition for `\int_to_arabic:n`. This function is documented on page 180.)

```

\int_to_symbols:nnn
\__int_to_symbols:nnnn
\__int_to_symbols:ennn

```

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

18704 \cs_new:Npn \int_to_symbols:nnn #1#2#3
18705 {
18706   \int_compare:nNnTF {#1} > {#2}
18707   {
18708     \__int_to_symbols:ennn
18709     {
18710       \int_case:nn
18711       { 1 + \int_mod:nn { #1 - 1 } {#2} }
18712       {#3}
18713     }
18714     {#1} {#2} {#3}
18715   }
18716   { \int_case:nn {#1} {#3} }
18717 }
18718 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
18719 {
18720   \exp_args:Nf \int_to_symbols:nnn
18721   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
18722   #1
18723 }
18724 \cs_generate_variant:Nn \__int_to_symbols:nnnn { e }

```

(End of definition for \int_to_symbols:nnn and __int_to_symbols:nnnn. This function is documented on page 181.)

```

\int_to_alph:n
\int_to_Alph:n

```

These both use the above function with input functions that make sense for the alphabet in English.

```

18725 \cs_new:Npn \int_to_alph:n #1
18726 {
18727   \int_to_symbols:nnn {#1} { 26 }
18728   {
18729     { 1 } { a }
18730     { 2 } { b }
18731     { 3 } { c }
18732     { 4 } { d }
18733     { 5 } { e }
18734     { 6 } { f }
18735     { 7 } { g }
18736     { 8 } { h }
18737     { 9 } { i }
18738     { 10 } { j }
18739     { 11 } { k }
18740     { 12 } { l }
18741     { 13 } { m }
18742     { 14 } { n }
18743     { 15 } { o }
18744     { 16 } { p }

```



```

18745         { 17 } { q }
18746         { 18 } { r }
18747         { 19 } { s }
18748         { 20 } { t }
18749         { 21 } { u }
18750         { 22 } { v }
18751         { 23 } { w }
18752         { 24 } { x }
18753         { 25 } { y }
18754         { 26 } { z }
18755     }
18756 }
18757 \cs_new:Npn \int_to_Alph:n #1
18758 {
18759     \int_to_symbols:nnn {#1} { 26 }
18760     {
18761         { 1 } { A }
18762         { 2 } { B }
18763         { 3 } { C }
18764         { 4 } { D }
18765         { 5 } { E }
18766         { 6 } { F }
18767         { 7 } { G }
18768         { 8 } { H }
18769         { 9 } { I }
18770         { 10 } { J }
18771         { 11 } { K }
18772         { 12 } { L }
18773         { 13 } { M }
18774         { 14 } { N }
18775         { 15 } { O }
18776         { 16 } { P }
18777         { 17 } { Q }
18778         { 18 } { R }
18779         { 19 } { S }
18780         { 20 } { T }
18781         { 21 } { U }
18782         { 22 } { V }
18783         { 23 } { W }
18784         { 24 } { X }
18785         { 25 } { Y }
18786         { 26 } { Z }
18787     }
18788 }

```

(End of definition for \int_to_alph:n and \int_to_Alph:n. These functions are documented on page 180.)

<pre> \int_to_base:nn \int_to_Base:nn __int_to_base:nn __int_to_Base:nnN __int_to_Base:nnN __int_to_base:nnnN __int_to_Base:nnnN __int_to_letter:n __int_to_Letter:n </pre>	<pre> 18789 \cs_new:Npn \int_to_base:nn #1 18790 { \exp_args:Nf __int_to_base:nn { \int_eval:n {#1} } } 18791 \cs_new:Npn \int_to_Base:nn #1 </pre> Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or \c_empty_tl, and feed the absolute value to the next auxiliary function.

```

18792 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
18793 \cs_new:Npn \__int_to_base:nn #1#2
18794 {
18795   \int_compare:nNnTF {#1} < 0
18796     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
18797     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
18798   }
18799 \cs_new:Npn \__int_to_Base:nn #1#2
18800 {
18801   \int_compare:nNnTF {#1} < 0
18802     { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
18803     { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
18804   }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

18805 \cs_new:Npn \__int_to_base:nnN #1#2#3
18806 {
18807   \int_compare:nNnTF {#1} < {#2}
18808     { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
18809     {
18810       \exp_args:Nf \__int_to_base:nnnN
18811         { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
18812         {#1}
18813         {#2}
18814         #3
18815     }
18816   }
18817 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
18818 {
18819   \exp_args:Nf \__int_to_base:nnN
18820     { \int_div_truncate:nn {#2} {#3} }
18821     {#3}
18822     #4
18823   #1
18824 }
18825 \cs_new:Npn \__int_to_Base:nnN #1#2#3
18826 {
18827   \int_compare:nNnTF {#1} < {#2}
18828     { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
18829     {
18830       \exp_args:Nf \__int_to_Base:nnnN
18831         { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
18832         {#1}
18833         {#2}
18834         #3
18835     }
18836   }
18837 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
18838 {

```

```

18839 \exp_args:Nf \__int_to_Base:nnN
18840 { \int_div_truncate:nn {#2} {#3} }
18841 {#3}
18842 #4
18843 #1
18844 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

18845 \cs_new:Npn \__int_to_letter:n #1
18846 {
18847   \exp_after:wN \exp_after:wN
18848   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
18849     a
18850   \or: b
18851   \or: c
18852   \or: d
18853   \or: e
18854   \or: f
18855   \or: g
18856   \or: h
18857   \or: i
18858   \or: j
18859   \or: k
18860   \or: l
18861   \or: m
18862   \or: n
18863   \or: o
18864   \or: p
18865   \or: q
18866   \or: r
18867   \or: s
18868   \or: t
18869   \or: u
18870   \or: v
18871   \or: w
18872   \or: x
18873   \or: y
18874   \or: z
18875   \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
18876   \fi:
18877 }
18878 \cs_new:Npn \__int_to_Letter:n #1
18879 {
18880   \exp_after:wN \exp_after:wN
18881   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
18882     A
18883   \or: B
18884   \or: C
18885   \or: D

```

```

18886 \or: E
18887 \or: F
18888 \or: G
18889 \or: H
18890 \or: I
18891 \or: J
18892 \or: K
18893 \or: L
18894 \or: M
18895 \or: N
18896 \or: O
18897 \or: P
18898 \or: Q
18899 \or: R
18900 \or: S
18901 \or: T
18902 \or: U
18903 \or: V
18904 \or: W
18905 \or: X
18906 \or: Y
18907 \or: Z
18908 \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
18909 \fi:
18910 }

```

(End of definition for \int_to_base:nn and others. These functions are documented on page 181.)

```

\int_to_bin:n  Wrappers around the generic function.
\int_to_hex:n  18911 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n  18912 { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n  18913 \cs_new:Npn \int_to_hex:n #1
                18914 { \int_to_base:nn {#1} { 16 } }
                18915 \cs_new:Npn \int_to_Hex:n #1
                18916 { \int_to_Base:nn {#1} { 16 } }
                18917 \cs_new:Npn \int_to_oct:n #1
                18918 { \int_to_base:nn {#1} { 8 } }

```

(End of definition for \int_to_bin:n and others. These functions are documented on page 181.)

```

\int_to_roman:n The \__int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n what is actually wanted is letters. The approach here is to convert the output of the
                  primitive into letters using appropriate control sequence names. That keeps everything
\__int_to_roman:N expandable. The loop is terminated by the conversion of the Q.
\__int_to_roman_i:w 18919 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman_v:w 18920 {
\__int_to_roman_x:w 18921 \exp_after:wN \__int_to_roman:N
\__int_to_roman_l:w 18922 \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_c:w 18923 }
\__int_to_roman_d:w 18924 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_m:w 18925 {
\__int_to_roman_Q:w 18926 \use:c { __int_to_roman_ #1 :w }
\__int_to_Roman_i:w 18927 \__int_to_roman:N
\__int_to_Roman_v:w 18928 }
\__int_to_Roman_x:w
\__int_to_Roman_l:w
\__int_to_Roman_c:w
\__int_to_Roman_d:w
\__int_to_Roman_m:w
\__int_to_Roman_Q:w

```

```

18929 \cs_new:Npn \int_to_Roman:n #1
18930 {
18931   \exp_after:wN \__int_to_Roman_aux:N
18932   \__int_to_roman:w \int_eval:n {#1} Q
18933 }
18934 \cs_new:Npn \__int_to_Roman_aux:N #1
18935 {
18936   \use:c { __int_to_Roman_ #1 :w }
18937   \__int_to_Roman_aux:N
18938 }
18939 \cs_new:Npn \__int_to_roman_i:w { i }
18940 \cs_new:Npn \__int_to_roman_v:w { v }
18941 \cs_new:Npn \__int_to_roman_x:w { x }
18942 \cs_new:Npn \__int_to_roman_l:w { l }
18943 \cs_new:Npn \__int_to_roman_c:w { c }
18944 \cs_new:Npn \__int_to_roman_d:w { d }
18945 \cs_new:Npn \__int_to_roman_m:w { m }
18946 \cs_new:Npn \__int_to_roman_Q:w #1 { }
18947 \cs_new:Npn \__int_to_Roman_i:w { I }
18948 \cs_new:Npn \__int_to_Roman_v:w { V }
18949 \cs_new:Npn \__int_to_Roman_x:w { X }
18950 \cs_new:Npn \__int_to_Roman_l:w { L }
18951 \cs_new:Npn \__int_to_Roman_c:w { C }
18952 \cs_new:Npn \__int_to_Roman_d:w { D }
18953 \cs_new:Npn \__int_to_Roman_m:w { M }
18954 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End of definition for `\int_to_roman:n` and others. These functions are documented on page 182.)

64.9 Converting from other formats to integers

`\__int_pass_signs:wn` Called as `\__int_pass_signs:wn <signs and digits> \s__int_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first non-sign token (and removes `\s__int_stop`). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```

18955 \cs_new:Npn \__int_pass_signs:wn #1
18956 {
18957   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
18958   \exp_after:wN \__int_pass_signs:wn
18959   \else:
18960     \exp_after:wN \__int_pass_signs_end:wn
18961     \exp_after:wN #1
18962   \fi:
18963 }
18964 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

```

(End of definition for `\__int_pass_signs:wn` and `\__int_pass_signs_end:wn`.)

`\int_from_alph:n` First take care of signs then loop through the input using the recursion quarks. The `\__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `\__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```

18965 \cs_new:Npn \int_from_alph:n #1

```

```

18966 {
18967   \int_eval:n
18968   {
18969     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
18970     \s__int_stop { \__int_from_alph:nN { 0 } }
18971     \q__int_recursion_tail \q__int_recursion_stop
18972   }
18973 }
18974 \cs_new:Npn \__int_from_alph:nN #1#2
18975 {
18976   \__int_if_recursion_tail_stop_do:Nn #2 {#1}
18977   \exp_args:Nf \__int_from_alph:nN
18978     { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
18979 }
18980 \cs_new:Npn \__int_from_alph:N #1
18981 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End of definition for `\int_from_alph:n`, `\__int_from_alph:nN`, and `\__int_from_alph:N`. This function is documented on page 182.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `\__int_from_base:nnN`. To convert a single character, `\__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

18982 \cs_new:Npn \int_from_base:nn #1#2
18983 {
18984   \int_eval:n
18985   {
18986     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
18987     \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
18988     \q__int_recursion_tail \q__int_recursion_stop
18989   }
18990 }
18991 \cs_new:Npn \__int_from_base:nnN #1#2#3
18992 {
18993   \__int_if_recursion_tail_stop_do:Nn #3 {#1}
18994   \exp_args:Nf \__int_from_base:nnN
18995     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
18996     {#2}
18997 }
18998 \cs_new:Npn \__int_from_base:N #1
18999 {
19000   \int_compare:nNnTF { '#1 } < { 58 }
19001     {#1}
19002     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
19003 }

```

(End of definition for `\int_from_base:nn`, `\__int_from_base:nnN`, and `\__int_from_base:N`. This function is documented on page 183.)

`\int_from_bin:n` Wrappers around the generic function.

```

19004 \cs_new:Npn \int_from_bin:n #1
19005 { \int_from_base:nn {#1} { 2 } }

```

```

19006 \cs_new:Npn \int_from_hex:n #1
19007   { \int_from_base:nn {#1} { 16 } }
19008 \cs_new:Npn \int_from_oct:n #1
19009   { \int_from_base:nn {#1} { 8 } }

```

(End of definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 182.)

`\c_int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

\c_int_from_roman_v_int 19010 \int_const:cn { c__int_from_roman_i_int } { 1 }
\c_int_from_roman_x_int 19011 \int_const:cn { c__int_from_roman_v_int } { 5 }
\c_int_from_roman_l_int 19012 \int_const:cn { c__int_from_roman_x_int } { 10 }
\c_int_from_roman_c_int 19013 \int_const:cn { c__int_from_roman_l_int } { 50 }
\c_int_from_roman_d_int 19014 \int_const:cn { c__int_from_roman_c_int } { 100 }
\c_int_from_roman_m_int 19015 \int_const:cn { c__int_from_roman_d_int } { 500 }
\c_int_from_roman_I_int 19016 \int_const:cn { c__int_from_roman_m_int } { 1000 }
\c_int_from_roman_V_int 19017 \int_const:cn { c__int_from_roman_I_int } { 1 }
\c_int_from_roman_X_int 19018 \int_const:cn { c__int_from_roman_V_int } { 5 }
\c_int_from_roman_L_int 19019 \int_const:cn { c__int_from_roman_X_int } { 10 }
\c_int_from_roman_C_int 19020 \int_const:cn { c__int_from_roman_L_int } { 50 }
\c_int_from_roman_D_int 19021 \int_const:cn { c__int_from_roman_C_int } { 100 }
\c_int_from_roman_M_int 19022 \int_const:cn { c__int_from_roman_D_int } { 500 }
19023 \int_const:cn { c__int_from_roman_M_int } { 1000 }

```

(End of definition for `\c__int_from_roman_i_int` and others.)

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by $\text{\TeX}$. If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by `-1`.

```

19024 \cs_new:Npn \int_from_roman:n #1
19025   {
19026     \int_eval:n
19027     {
19028       (
19029         0
19030         \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
19031         \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
19032       )
19033     }
19034   }
19035 \cs_new:Npn \__int_from_roman:NN #1#2
19036   {
19037     \__int_if_recursion_tail_stop:N #1
19038     \int_if_exist:cF { c__int_from_roman_ #1 _int }
19039     { \__int_from_roman_error:w }
19040     \__int_if_recursion_tail_stop_do:Nn #2
19041     { + \use:c { c__int_from_roman_ #1 _int } }
19042     \int_if_exist:cF { c__int_from_roman_ #2 _int }
19043     { \__int_from_roman_error:w }
19044     \int_compare:nNnTF
19045     { \use:c { c__int_from_roman_ #1 _int } }
19046     <
19047     { \use:c { c__int_from_roman_ #2 _int } }

```

```

19048     {
19049         + \use:c { c__int_from_roman_ #2 _int }
19050         - \use:c { c__int_from_roman_ #1 _int }
19051         \__int_from_roman:NN
19052     }
19053     {
19054         + \use:c { c__int_from_roman_ #1 _int }
19055         \__int_from_roman:NN #2
19056     }
19057 }
19058 \cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
19059 { #2 * 0 - 1 }

```

(End of definition for `\int_from_roman:n`, `\__int_from_roman:NN`, and `\__int_from_roman_error:w`. This function is documented on page 183.)

64.10 Viewing integer

`\int_show:N` Diagnostics.
`\int_show:c` 19060 `\cs_new_eq:NN \int_show:N \__kernel_register_show:N`
`\__int_show:nN` 19061 `\cs_generate_variant:Nn \int_show:N { c }`

(End of definition for `\int_show:N` and `\__int_show:nN`. This function is documented on page 183.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

19062 \cs_new_protected:Npn \int_show:n
19063 { \__kernel_msg_show_eval:Nn \int_eval:n }

```

(End of definition for `\int_show:n`. This function is documented on page 183.)

`\int_log:N` Diagnostics.
`\int_log:c` 19064 `\cs_new_eq:NN \int_log:N \__kernel_register_log:N`
19065 `\cs_generate_variant:Nn \int_log:N { c }`

(End of definition for `\int_log:N`. This function is documented on page 183.)

`\int_log:n` Similar to `\int_show:n`.

```

19066 \cs_new_protected:Npn \int_log:n
19067 { \__kernel_msg_log_eval:Nn \int_eval:n }

```

(End of definition for `\int_log:n`. This function is documented on page 183.)

64.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End of definition for `\int_rand:nn`. This function is documented on page 183.)

64.12 Constant integers

\c_zero_int The zero is defined in l3basics.

\c_one_int 19068 \int_const:Nn \c_one_int { 1 }

(End of definition for \c_zero_int and \c_one_int. These variables are documented on page 184.)

\c_max_int The largest number allowed is $2^{31} - 1$

19069 \int_const:Nn \c_max_int { 2 147 483 647 }

(End of definition for \c_max_int. This variable is documented on page 184.)

\c_max_char_int The largest character code is 1114111 (hexadecimal 10FFFF) in XeTeX and LuaTeX and 255 in other engines. In many places pTeX and upTeX support larger character codes but for instance the values of \lccode are restricted to $[0, 255]$.

```
19070 \int_const:Nn \c_max_char_int
19071 {
19072   \if_int_odd:w 0
19073     \cs_if_exist:NT \tex luatexversion:D { 1 }
19074     \cs_if_exist:NT \tex XeTeXversion:D { 1 } ~
19075     "10FFFF
19076   \else:
19077     "FF
19078   \fi:
19079 }
```

(End of definition for \c_max_char_int. This variable is documented on page 184.)

64.13 Scratch integers

\l_tmpa_int We provide two local and two global scratch counters, maybe we need more or less.

\l_tmpb_int 19080 \int_new:N \l_tmpa_int

\g_tmpa_int 19081 \int_new:N \l_tmpb_int

\g_tmpb_int 19082 \int_new:N \g_tmpa_int

19083 \int_new:N \g_tmpb_int

(End of definition for \l_tmpa_int and others. These variables are documented on page 184.)

64.14 Integers for earlier modules

<@@=seq>

\l__int_tmpa_int

\l__int_tmpb_int

19084 \int_new:N \l__int_tmpa_int

19085 \int_new:N \l__int_tmpb_int

(End of definition for \l__int_tmpa_int and \l__int_tmpb_int.)

19086 </code>

Chapter 65

l3flag implementation

```
19087 < *code >
19088 < @@=flag >
      The following test files are used for this code: m3flag001.
__flag_sep:
19089 \cs_new_eq:NN __flag_sep: \__kernel_int_sep:
      (End of definition for __flag_sep:.)
```

65.1 Protected flag commands

The height h of a flag (which is initially zero) is stored by setting control sequences of the form $\backslash\langle flag\ name\rangle\langle integer\rangle$ to $\backslash relax$ for $0 \leq \langle integer\rangle < h$. These control sequences are produced by $\backslash cs:w\langle flag\ var\rangle\langle integer\rangle\backslash cs_end:$, namely the $\langle flag\ var\rangle$ is actually a (protected) macro expanding to its own csname.

```
\flag_new:N Evaluate the csname of #1 for use in constructing the various indexed macros.
\flag_new:c
19090 \cs_new_protected:Npn \flag_new:N #1
19091 { \cs_new_protected:Npe #1 { \cs_to_str:N #1 } }
19092 \cs_generate_variant:Nn \flag_new:N { c }
      (End of definition for \flag_new:N. This function is documented on page 187.)
```

```
\l_tmpa_flag Two flag variables for scratch use.
\l_tmpb_flag
19093 \flag_new:N \l_tmpa_flag
19094 \flag_new:N \l_tmpb_flag
      (End of definition for \l_tmpa_flag and \l_tmpb_flag. These variables are documented on page 189.)
```

```
\flag_clear:N Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined
\flag_clear:c control sequence. We don't use \cs_undefine:c because that would act globally.
__flag_clear:wN
19095 \cs_new_protected:Npn \flag_clear:N #1
19096 {
19097   __flag_clear:wN 0 __flag_sep: #1
19098   \prg_break_point:
19099 }
19100 \cs_generate_variant:Nn \flag_clear:N { c }
```

```

19101 \cs_new_protected:Npn \__flag_clear:wN #1 \__flag_sep: #2
19102 {
19103   \if_cs_exist:w #2 #1 \cs_end: \else:
19104     \prg_break:n
19105   \fi:
19106   \cs_set_eq:cN { #2 #1 } \tex_undefined:D
19107   \exp_after:wN \__flag_clear:wN
19108   \int_value:w \int_eval:w \c_one_int + #1 \__flag_sep: #2
19109 }

```

(End of definition for `\flag_clear:N` and `\__flag_clear:wN`. This function is documented on page 188.)

`\flag_clear_new:N` As for other datatypes, clear the `(flag var)` or create a new one, as appropriate.

```

\flag_clear_new:c 19110 \cs_new_protected:Npn \flag_clear_new:N #1
19111 { \flag_if_exist:NTF #1 { \flag_clear:N } { \flag_new:N } #1 }
19112 \cs_generate_variant:Nn \flag_clear_new:N { c }

```

(End of definition for `\flag_clear_new:N`. This function is documented on page 188.)

`\flag_show:N` Show the height (terminal or log file) using appropriate `l3msg` auxiliaries.

```

\flag_show:c 19113 \cs_new_protected:Npn \flag_show:N { \__flag_show:NN \tl_show:n }
\flag_log:N 19114 \cs_generate_variant:Nn \flag_show:N { c }
\flag_log:c 19115 \cs_new_protected:Npn \flag_log:N { \__flag_show:NN \tl_log:n }
\__flag_show:NN 19116 \cs_generate_variant:Nn \flag_log:N { c }
19117 \cs_new_protected:Npn \__flag_show:NN #1#2
19118 {
19119   \__kernel_chk_defined:NT #2
19120   { \exp_args:Ne #1 { \tl_to_str:n { #2 height } = \flag_height:N #2 } }
19121 }

```

(End of definition for `\flag_show:N`, `\flag_log:N`, and `\__flag_show:NN`. These functions are documented on page 188.)

65.2 Expandable flag commands

`\flag_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\flag_if_exist_p:c 19122 \prg_new_eq_conditional:Nnn \flag_if_exist:N \cs_if_exist:N
\flag_if_exist:NTF 19123 { TF , T , F , p }
\flag_if_exist:cTF 19124 \prg_new_eq_conditional:Nnn \flag_if_exist:c \cs_if_exist:c
19125 { TF , T , F , p }

```

(End of definition for `\flag_if_exist:NTF`. This function is documented on page 188.)

`\flag_if_raised_p:N` Test if the flag has a non-zero height, by checking the 0 control sequence.

```

\flag_if_raised_p:c 19126 \prg_new_conditional:Npnn \flag_if_raised:N #1 { p , T , F , TF }
\flag_if_raised:NTF 19127 {
\flag_if_raised:cTF 19128   \if_cs_exist:w #1 0 \cs_end:
19129     \prg_return_true:
19130   \else:
19131     \prg_return_false:
19132   \fi:
19133 }
19134 \prg_generate_conditional_variant:Nnn \flag_if_raised:N
19135 { c } { p , T , F , TF }

```

(End of definition for `\flag_if_raised:NTF`. This function is documented on page 188.)

```

\flag_height:N Extract the value of the flag by going through all of the control sequences starting from
\flag_height:c 0.
\__flag_height_loop:wN 19136 \cs_new:Npn \flag_height:N #1 { \__flag_height_loop:wN 0 \__flag_sep: #1 }
\__flag_height_end:wN 19137 \cs_new:Npn \__flag_height_loop:wN #1 \__flag_sep: #2
19138 {
19139   \if_cs_exist:w #2 #1 \cs_end: \else:
19140     \exp_after:wN \__flag_height_end:wN
19141   \fi:
19142   \exp_after:wN \__flag_height_loop:wN
19143   \int_value:w \int_eval:w \c_one_int + #1 \__flag_sep: #2
19144 }
19145 \cs_new:Npn \__flag_height_end:wN #1 + #2 \__flag_sep: #3 {#2}
19146 \cs_generate_variant:Nn \flag_height:N { c }

```

(End of definition for `\flag_height:N`, `\__flag_height_loop:wN`, and `\__flag_height_end:wN`. This function is documented on page 188.)

```

\flag_raise:N Change the appropriate control sequence to \relax by expanding a \cs:w ... \cs_end:
\flag_raise:c construction, then pass it to \use_none:n to avoid leaving anything in the input stream.
19147 \cs_new:Npn \flag_raise:N #1
19148 { \exp_after:wN \use_none:n \cs:w #1 \flag_height:N #1 \cs_end: }
19149 \cs_generate_variant:Nn \flag_raise:N { c }

```

(End of definition for `\flag_raise:N`. This function is documented on page 188.)

```

\flag_ensure_raised:N Pass the control sequence with name <flag name>0 to \use_none:n. Constructing the
\flag_ensure_raised:c control sequence ensures that it changes from being undefined (if it was so) to being
\relax.

```

```

19150 \cs_new:Npn \flag_ensure_raised:N #1
19151 { \exp_after:wN \use_none:n \cs:w #1 0 \cs_end: }
19152 \cs_generate_variant:Nn \flag_ensure_raised:N { c }

```

(End of definition for `\flag_ensure_raised:N`. This function is documented on page 188.)

65.3 Old n-type flag commands

Here we keep the old flag commands since our policy is to no longer delete deprecated functions. The idea is to simply map `<flag name>` to `\l_<flag name>_flag`. When the debugging code is activated, it checks existence of the N-type flag variables that result.

```

\flag_new:n
\flag_clear:n 19153 \cs_new_protected:Npn \flag_new:n #1 { \flag_new:c { l_#1_flag } }
\flag_clear_new:n 19154 \cs_new_protected:Npn \flag_clear:n #1 { \flag_clear:c { l_#1_flag } }
\flag_if_exist_p:n 19155 \cs_new_protected:Npn \flag_clear_new:n #1 { \flag_clear_new:c { l_#1_flag } }
\flag_if_exist:nTF 19156 \cs_new:Npn \flag_if_exist_p:n #1 { \flag_if_exist_p:c { l_#1_flag } }
\flag_if_raised_p:n 19157 \cs_new:Npn \flag_if_exist:nTF #1 { \flag_if_exist:cT { l_#1_flag } }
\flag_if_raised:nTF 19158 \cs_new:Npn \flag_if_exist:nF #1 { \flag_if_exist:cF { l_#1_flag } }
\flag_height:n 19159 \cs_new:Npn \flag_if_exist:nTF #1 { \flag_if_exist:cTF { l_#1_flag } }
\flag_raise:n 19160 \cs_new:Npn \flag_if_raised_p:n #1 { \flag_if_raised_p:c { l_#1_flag } }
\flag_ensure_raised:n 19161 \cs_new:Npn \flag_if_raised:nT #1 { \flag_if_raised:cT { l_#1_flag } }
19162 \cs_new:Npn \flag_if_raised:nF #1 { \flag_if_raised:cF { l_#1_flag } }

```

```

19163 \cs_new:Npn \flag_if_raised:nTF #1 { \flag_if_raised:cTF { l_#1_flag } }
19164 \cs_new:Npn \flag_height:n #1 { \flag_height:c { l_#1_flag } }
19165 \cs_new:Npn \flag_raise:n #1 { \flag_raise:c { l_#1_flag } }
19166 \cs_new:Npn \flag_ensure_raised:n #1 { \flag_ensure_raised:c { l_#1_flag } }

```

(End of definition for \flag_new:n and others.)

```

\flag_show:n To avoid changing the output here we mostly keep the old code.
\flag_log:n
\__flag_show:Nn
19167 \cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
19168 \cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
19169 \cs_new_protected:Npn \__flag_show:Nn #1#2
19170 {
19171   \exp_args:Nc \__kernel_chk_defined:NT { l_#2_flag }
19172   {
19173     \exp_args:Ne #1
19174     { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
19175   }
19176 }

```

(End of definition for \flag_show:n, \flag_log:n, and __flag_show:Nn.)

```

19177 </code>

```

Chapter 66

l3clist implementation

The following test files are used for this code: m3clist002.

```
19178 <*code>
19179 <@@=clist>
```

\c_empty_clist An empty comma list is simply an empty token list.

```
19180 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End of definition for \c_empty_clist. This variable is documented on page 200.)

\l__clist_tmp_clist Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before \clist_new:N

```
19181 \tl_new:N \l__clist_tmp_clist
```

(End of definition for \l__clist_tmp_clist.)

\s__clist_mark Internal scan marks.

```
\s__clist_stop 19182 \scan_new:N \s__clist_mark
19183 \scan_new:N \s__clist_stop
```

(End of definition for \s__clist_mark and \s__clist_stop.)

__clist_use_none_delimit_by_s_mark:w Functions to gobble up to a scan mark.

```
\__clist_use_none_delimit_by_s_stop:w 19184 \cs_new:Npn \__clist_use_none_delimit_by_s_mark:w #1 \s__clist_mark { }
\__clist_use_i_delimit_by_s_stop:nw 19185 \cs_new:Npn \__clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }
19186 \cs_new:Npn \__clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}
```

(End of definition for __clist_use_none_delimit_by_s_mark:w, __clist_use_none_delimit_by_s_stop:w, and __clist_use_i_delimit_by_s_stop:nw.)

__clist_tmp:w A temporary function for various purposes.

```
19187 \cs_new_protected:Npn \__clist_tmp:w { }
```

(End of definition for __clist_tmp:w.)

66.1 Removing spaces around items

`\__clist_trim_next:w` Called as `\exp:w \__clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

19188 \cs_new:Npn \__clist_trim_next:w #1 ,
19189 {
19190     \tl_if_empty:oTF { \use_none:nn #1 ? }
19191     { \__clist_trim_next:w \prg_do_nothing: }
19192     { \tl_trim_spaces_apply:oN {#1} \exp_end: }
19193 }
```

(End of definition for `\__clist_trim_next:w`.)

`\__clist_sanitize:n` The auxiliary `\__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `\__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

`\__clist_sanitize:Nn`

```

19194 \cs_new:Npn \__clist_sanitize:n #1
19195 {
19196     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
19197     \exp:w \__clist_trim_next:w \prg_do_nothing:
19198     #1 , \s__clist_stop \prg_break: , \prg_break_point:
19199 }
19200 \cs_new:Npn \__clist_sanitize:Nn #1#2
19201 {
19202     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
19203     #1 \__clist_wrap_item:w #2 ,
19204     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
19205     \exp:w \__clist_trim_next:w \prg_do_nothing:
19206 }
```

(End of definition for `\__clist_sanitize:n` and `\__clist_sanitize:Nn`.)

`\__clist_if_wrap:nTF` True if the argument must be wrapped to avoid getting altered by some clist operations.
`\__clist_if_wrap:w` That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

If the argument starts or ends with a space or contains a comma then one of the three arguments of `\__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `\__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

19207 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
```

```

19208 {
19209   \tl_if_empty:oTF
19210   {
19211     \__clist_if_wrap:w
19212     \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
19213     \s__clist_mark , ~ \s__clist_mark #1 ,
19214   }
19215   {
19216     \tl_if_head_is_group:nTF { #1 { } }
19217     {
19218       \tl_if_empty:nTF {#1}
19219       { \prg_return_true: }
19220       {
19221         \tl_if_empty:oTF { \use_none:n #1}
19222         { \prg_return_true: }
19223         { \prg_return_false: }
19224       }
19225     }
19226     { \prg_return_false: }
19227   }
19228   { \prg_return_true: }
19229 }
19230 \cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

```

(End of definition for __clist_if_wrap:nTF and __clist_if_wrap:w.)

__clist_wrap_item:w Safe items are put in \exp_not:n, otherwise we put an extra set of braces.

```

19231 \cs_new:Npn \__clist_wrap_item:w #1 ,
19232 { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End of definition for __clist_wrap_item:w.)

66.2 Allocation and initialization

\clist_new:N Internally, comma lists are just token lists.

```

\clist_new:c 19233 \cs_new_eq:NN \clist_new:N \tl_new:N
19234 \cs_new_eq:NN \clist_new:c \tl_new:c

```

(End of definition for \clist_new:N. This function is documented on page 191.)

\clist_const:Nn Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:Ne 19235 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:Nx 19236 { \tl_const:Ne #1 { \__clist_sanitize:n {#2} } }
\clist_const:cn 19237 \cs_generate_variant:Nn \clist_const:Nn { Ne , c , ce }
\clist_const:ce 19238 \cs_generate_variant:Nn \clist_const:Nn { Nx , cx }
\clist_const:cx

```

(End of definition for \clist_const:Nn. This function is documented on page 191.)

\clist_clear:N Clearing comma lists is just the same as clearing token lists.

```

\clist_clear:c 19239 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N 19240 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c 19241 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
19242 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c

```


(End of definition for \clist_clear:N and \clist_gclear:N. These functions are documented on page 191.)

```
\clist_clear_new:N Once again a copy from the token list functions.
\clist_clear_new:c 19243 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 19244 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 19245 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
19246 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(End of definition for \clist_clear_new:N and \clist_gclear_new:N. These functions are documented on page 191.)

```
\clist_set_eq:NN Once again, these are simple copies from the token list functions.
\clist_set_eq:cN 19247 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 19248 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 19249 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 19250 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 19251 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 19252 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 19253 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
19254 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End of definition for \clist_set_eq:NN and \clist_gset_eq:NN. These functions are documented on page 191.)

```
\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. Safe
\clist_set_from_seq:cN items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma
\clist_set_from_seq:Nc must be removed, except in the case of an empty comma-list.
\clist_set_from_seq:cc 19255 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:NN 19256 { \__clist_set_from_seq:NNNN \clist_clear:N \__kernel_tl_set:Nx }
\clist_gset_from_seq:cN 19257 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:Nc 19258 { \__clist_set_from_seq:NNNN \clist_gclear:N \__kernel_tl_gset:Nx }
\clist_gset_from_seq:cc 19259 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
\__clist_set_from_seq:NNNN 19260 {
\__clist_set_from_seq:n 19261 \seq_if_empty:NTF #4
19262 { #1 #3 }
19263 {
19264 #2 #3
19265 {
19266 \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
19267 \seq_map_function:NN #4 \__clist_set_from_seq:n
19268 }
19269 }
19270 }
19271 \cs_new:Npn \__clist_set_from_seq:n #1
19272 {
19273 ,
19274 \__clist_if_wrap:NTF {#1}
19275 { \exp_not:n { {#1} } }
19276 { \exp_not:n {#1} }
19277 }
19278 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
19279 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
19280 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
19281 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(End of definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 191.)

```
\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
\__clist_concat:NNNN
```

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
19282 \cs_new_protected:Npn \clist_concat:NNN
19283   { \__clist_concat:NNNN \__kernel_tl_set:Nx }
19284 \cs_new_protected:Npn \clist_gconcat:NNN
19285   { \__clist_concat:NNNN \__kernel_tl_gset:Nx }
19286 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
19287   {
19288     #1 #2
19289     {
19290       \exp_not:o #3
19291       \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
19292       \exp_not:o #4
19293     }
19294   }
19295 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
19296 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(End of definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `\__clist_concat:NNNN`. These functions are documented on page 192.)

```
\clist_if_exist_p:N
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF
```

Copies of the `cs` functions defined in `l3basics`.

```
19297 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
19298   { TF , T , F , p }
19299 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
19300   { TF , T , F , p }
```

(End of definition for `\clist_if_exist:NTF`. This function is documented on page 192.)

66.3 Adding data to comma lists

```
\clist_set:Nn
```

```
\clist_set:NV
\clist_set:Nv
\clist_set:Ne
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:cv
\clist_set:ce
\clist_set:co
```

```
19301 \cs_new_protected:Npn \clist_set:Nn #1#2
19302   { \__kernel_tl_set:Nx #1 { \__clist_sanitize:n {#2} } }
19303 \cs_new_protected:Npn \clist_gset:Nn #1#2
19304   { \__kernel_tl_gset:Nx #1 { \__clist_sanitize:n {#2} } }
19305 \cs_generate_variant:Nn \clist_set:Nn { NV , Nv , Ne , c , cV , cv , ce }
19306 \cs_generate_variant:Nn \clist_set:Nn { No , Nx , co , cx }
19307 \cs_generate_variant:Nn \clist_gset:Nn { NV , Nv , Ne , c , cV , cv , ce }
19308 \cs_generate_variant:Nn \clist_gset:Nn { No , Nx , co , cx }
```

(End of definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 192.)

```
\clist_set:cx
```

```
\clist_put_left:Nn
\clist_gset:Nn
\clist_put_left:Nv
\clist_gset:Nv
\clist_put_left:Ne
\clist_gset:Ne
\clist_put_left:No
\clist_gset:No
\clist_put_left:Nx
\clist_gset:Nx
\clist_put_left:cn
\clist_gset:cn
\clist_put_left:cV
\clist_gset:cV
\clist_put_left:cv
\clist_gset:cv
\clist_put_left:ce
\clist_gset:ce
\clist_put_left:co
\clist_gset:co
\clist_put_left:cx
\clist_gset:cx
\clist_gput_left:Nn
```

Everything is based on concatenation after storing in `\l__clist_tmp_clist`. This avoids having to worry here about space-trimming and so on.

```
19309 \cs_new_protected:Npn \clist_put_left:Nn
19310   { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
19311 \cs_new_protected:Npn \clist_gput_left:Nn
19312   { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
19313 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
```

```

19314 {
19315     #2 \l__clist_tmp_clist {#4}
19316     #1 #3 \l__clist_tmp_clist #3
19317 }
19318 \cs_generate_variant:Nn \clist_put_left:Nn { NV , Nv , Ne , c , cV , cv , ce }
19319 \cs_generate_variant:Nn \clist_put_left:Nn { No , Nx , co , cx }
19320 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , Nv , Ne , c , cV , cv , ce }
19321 \cs_generate_variant:Nn \clist_gput_left:Nn { No , Nx , co , cx }

```

(End of definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `\__clist_put_left:NNNn`. These functions are documented on page 192.)

```

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:Nv
\clist_put_right:Ne
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:cv
\clist_put_right:ce
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:Nv
\clist_gput_right:Ne
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:cv
\clist_gput_right:ce
\clist_gput_right:cx
\clist_get:Nn
\clist_get:NV
\clist_get:Nv
\clist_get:Ne
\clist_get:No
\clist_get:Nx
\clist_get:cn
\clist_get:cV
\clist_get:cv
\clist_get:ce
\clist_get:cx
\__clist_put_right:NNNn
\__clist_get:wN

```

```

19322 \cs_new_protected:Npn \clist_put_right:Nn
19323 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
19324 \cs_new_protected:Npn \clist_gput_right:Nn
19325 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
19326 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
19327 {
19328     #2 \l__clist_tmp_clist {#4}
19329     #1 #3 #3 \l__clist_tmp_clist
19330 }
19331 \cs_generate_variant:Nn \clist_put_right:Nn
19332 { NV , Nv , Ne , c , cV , cv , ce }
19333 \cs_generate_variant:Nn \clist_put_right:Nn
19334 { No , Nx , co , cx }
19335 \cs_generate_variant:Nn \clist_gput_right:Nn
19336 { NV , Nv , Ne , c , cV , cv , ce }
19337 \cs_generate_variant:Nn \clist_gput_right:Nn
19338 { No , Nx , co , cx }

```

(End of definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `\__clist_put_right:NNNn`. These functions are documented on page 192.)

66.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

19339 \cs_new_protected:Npn \clist_get:NN #1#2
19340 {
19341     \if_meaning:w #1 \c_empty_clist
19342     \tl_set:Nn #2 { \q_no_value }
19343     \else:
19344         \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
19345     \fi:
19346 }
19347 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \s__clist_stop #3
19348 { \tl_set:Nn #3 {#1} }
19349 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End of definition for `\clist_get:NN` and `\__clist_get:wN`. This function is documented on page 198.)

`\clist_pop:NN` An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\__clist_pop:wwNNN` is a comma list ending in a comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```

19350 \cs_new_protected:Npn \clist_pop:NN
19351 { \__clist_pop:NNN \__kernel_tl_set:Nx }
19352 \cs_new_protected:Npn \clist_gpop:NN
19353 { \__clist_pop:NNN \__kernel_tl_gset:Nx }
19354 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
19355 {
19356   \if_meaning:w #2 \c_empty_clist
19357     \tl_set:Nn #3 { \q_no_value }
19358   \else:
19359     \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
19360   \fi:
19361 }
19362 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
19363 {
19364   \tl_set:Nn #5 {#1}
19365   #3 #4
19366   {
19367     \__clist_pop:wN \prg_do_nothing:
19368     #2 \exp_not:o
19369     , \s__clist_mark \use_none:n
19370     \s__clist_stop
19371   }
19372 }
19373 \cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
19374 \cs_generate_variant:Nn \clist_pop:NN { c }
19375 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End of definition for `\clist_pop:NN` and others. These functions are documented on page 198.)

`\clist_get:NNTF` The same, as branching code: very similar to the above.

```

19376 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
19377 {
19378   \if_meaning:w #1 \c_empty_clist
19379     \prg_return_false:
19380   \else:
19381     \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
19382     \prg_return_true:
19383   \fi:
19384 }
19385 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
19386 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
19387 { \__clist_pop_TF:NNN \__kernel_tl_set:Nx #1 #2 }
19388 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
19389 { \__clist_pop_TF:NNN \__kernel_tl_gset:Nx #1 #2 }
19390 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
19391 {
19392   \if_meaning:w #2 \c_empty_clist
19393     \prg_return_false:
19394   \else:

```

```

19395         \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
19396         \prg_return_true:
19397     \fi:
19398 }
19399 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
19400 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End of definition for \clist_get:NNTF and others. These functions are documented on page 198.)

\clist_push:Nn Pushing to a comma list is the same as adding on the left.

```

\clist_push:NV 19401 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No 19402 \cs_generate_variant:Nn \clist_push:Nn { NV , No , Nx , c , cV , co , cx }
\clist_push:Nx 19403 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_push:cn 19404 \cs_generate_variant:Nn \clist_gpush:Nn { NV , No , Nx , c , cV , co , cx }
\clist_push:cV
\clist_push:co
\clist_push:cx

```

(End of definition for \clist_push:Nn and \clist_gpush:Nn. These functions are documented on page 198.)

\clist_gpush:Nn

\clist_gpush:NV

\clist_gpush:No

\clist_gpush:Nx

\clist_gpush:cn

\clist_gpush:cV

\clist_gpush:co

\clist_gpush:cx

66.5 Modifying comma lists

An internal comma list and a sequence for the removal routines.

```

\l__clist_remove_clist:N 19405 \clist_new:N \l__clist_remove_clist
\l__clist_remove_seq:N 19406 \seq_new:N \l__clist_remove_seq

```

(End of definition for \l__clist_remove_clist and \l__clist_remove_seq.)

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.

```

\clist_remove_duplicates:c 19407 \cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N 19408 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c 19409 \cs_new_protected:Npn \clist_gremove_duplicates:N
\__clist_remove_duplicates:NN 19410 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
19411 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
19412 {
19413     \clist_clear:N \l__clist_remove_clist
19414     \clist_map_inline:Nn #2
19415     {
19416         \clist_if_in:NnF \l__clist_remove_clist {##1}
19417         {
19418             \tl_put_right:Ne \l__clist_remove_clist
19419             {
19420                 \clist_if_empty:NF \l__clist_remove_clist { , }
19421                 \__clist_if_wrap:nTF {##1} { \exp_not:n { {##1} } } { \exp_not:n { {##1} } }
19422             }
19423         }
19424     }
19425     #1 #2 \l__clist_remove_clist
19426 }
19427 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
19428 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```

(End of definition for \clist_remove_duplicates:N, \clist_gremove_duplicates:N, and __clist_remove_duplicates:NN. These functions are documented on page 193.)

```

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_remove_all:NV
\clist_remove_all:cV
\clist_remove_all:Ne
\clist_remove_all:ce
\clist_gremove_all:Nn
\clist_gremove_all:cn
\clist_gremove_all:NV
\clist_gremove_all:cV
\clist_gremove_all:Ne
\clist_gremove_all:ce
__clist_remove_all:NNNn
__clist_remove_all:w
__clist_remove_all:

```

The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `l3seq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `__clist_tmp:w` contains `\s__clist_mark`: in that case, `__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `__clist_tmp:w`), and lets `__clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```

19429 \cs_new_protected:Npn \clist_remove_all:Nn
19430 { __clist_remove_all:NNNn \clist_set_from_seq:NN __kernel_tl_set:Nx }
19431 \cs_new_protected:Npn \clist_gremove_all:Nn
19432 { __clist_remove_all:NNNn \clist_gset_from_seq:NN __kernel_tl_gset:Nx }
19433 \cs_new_protected:Npn __clist_remove_all:NNNn #1#2#3#4
19434 {
19435   __clist_if_wrap:nTF {#4}
19436   {
19437     \seq_set_from_clist:NN \l__clist_remove_seq #3
19438     \seq_remove_all:Nn \l__clist_remove_seq {#4}
19439     #1 #3 \l__clist_remove_seq
19440   }
19441   {
19442     \cs_set:Npn __clist_tmp:w ##1 , #4 ,
19443     {
19444       ##1
19445       , \s__clist_mark , __clist_use_none_delimit_by_s_stop:w ,
19446       __clist_remove_all:
19447     }
19448     #2 #3
19449     {
19450       \exp_after:wN __clist_remove_all:
19451       #3 , \s__clist_mark , #4 , \s__clist_stop
19452     }
19453     \clist_if_empty:NF #3
19454     {
19455       #2 #3
19456       {
19457         \exp_args:No \exp_not:o
19458         { \exp_after:wN \use_none:n #3 }
19459       }
19460     }

```

```

19461     }
19462   }
19463   \cs_new:Npn \__clist_remove_all:
19464     { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
19465   \cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
19466   \cs_generate_variant:Nn \clist_remove_all:Nn { c , NV , cV , Ne , ce }
19467   \cs_generate_variant:Nn \clist_gremove_all:Nn { c , NV , cV , Ne , ce }

```

(End of definition for `\clist_remove_all:Nn` and others. These functions are documented on page 193.)

`\clist_reverse:N` Use `\clist_reverse:n` in an e-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

```

\clist_reverse:c
\clist_greverse:N
\clist_greverse:c
19468 \cs_new_protected:Npn \clist_reverse:N #1
19469   { \__kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
19470 \cs_new_protected:Npn \clist_greverse:N #1
19471   { \__kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
19472 \cs_generate_variant:Nn \clist_reverse:N { c }
19473 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End of definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 193.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “`<item>`,”. We start from a comma list “`<item1>, ..., <itemn>`”. During the loop, the auxiliary `\__clist_reverse:wwNww` receives “`?<itemi>`” as #1, “`<itemi+1>, ..., <itemn>`” as #2, `\__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “`<itemi-1>, ..., <item1>`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `\__clist_reverse:wwNww` receives “`\s__clist_mark \__clist_reverse:wwNww !`” as its argument #1, thus `\__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

19474 \cs_new:Npn \clist_reverse:n #1
19475   {
19476     \__clist_reverse:wwNww ? #1 ,
19477     \s__clist_mark \__clist_reverse:wwNww ! ,
19478     \s__clist_mark \__clist_reverse_end:ww
19479     \s__clist_stop ? \s__clist_mark
19480   }
19481 \cs_new:Npn \__clist_reverse:wwNww
19482   #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
19483   { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
19484 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
19485   { \exp_not:o { \use_none:n #2 } }

```

(End of definition for `\clist_reverse:n`, `\__clist_reverse:wwNww`, and `\__clist_reverse_end:ww`. This function is documented on page 193.)

`\clist_sort:Nn` Implemented in `l3sort`.

`\clist_sort:cn`

`\clist_gsort:Nn` (End of definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 193.)

`\clist_gsort:cn`

66.6 Comma list conditionals

```

\clist_if_empty_p:N Simple copies from the token list variable material.
\clist_if_empty_p:c 19486 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
\clist_if_empty:N\TF 19487 { p , T , F , TF }
\clist_if_empty:c\TF 19488 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
19489 { p , T , F , TF }

```

(End of definition for \clist_if_empty:N\TF. This function is documented on page 194.)

```

\clist_if_empty_p:n As usual, we insert a token (here ?) before grabbing any argument: this avoids losing
\clist_if_empty:n\TF braces. The argument of \tl_if_empty:oTF is empty if #1 is ? followed by blank spaces
\__clist_if_empty_n:w (besides, this particular variant of the emptiness test is optimized). If the item of the
\__clist_if_empty_n:wNw comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second
auxiliary grabs \prg_return_false: as #2, unless every item in the comma list was blank
and the loop actually got broken by the trailing \s__clist_mark \prg_return_false:
item.

```

```

19490 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
19491 {
19492   \__clist_if_empty_n:w ? #1
19493   , \s__clist_mark \prg_return_false:
19494   , \s__clist_mark \prg_return_true:
19495   \s__clist_stop
19496 }
19497 \cs_new:Npn \__clist_if_empty_n:w #1 ,
19498 {
19499   \tl_if_empty:oTF { \use_none:nn #1 ? }
19500   { \__clist_if_empty_n:w ? }
19501   { \__clist_if_empty_n:wNw }
19502 }
19503 \cs_new:Npn \__clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}

```

(End of definition for \clist_if_empty:n\TF, __clist_if_empty_n:w, and __clist_if_empty_n:wNw. This function is documented on page 194.)

```

\clist_if_in:Nn\TF For “safe” items, we simply surround the comma list, and the item, with commas, then
\clist_if_in:NV\TF use the same code as for \tl_if_in:Nn. For “unsafe” items we follow the same route as
\clist_if_in:No\TF \seq_if_in:Nn, mapping through the list a comparison function. If found, return true
\clist_if_in:cn\TF and remove \prg_return_false:.
\clist_if_in:cV\TF 19504 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
\clist_if_in:co\TF 19505 {
\clist_if_in:nn\TF 19506   \exp_args:No \__clist_if_in_return:nnN #1 {#2} #1
\clist_if_in:nV\TF 19507 }
\clist_if_in:no\TF 19508 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
\__clist_if_in_return:nnN 19509 {
19510   \clist_set:Nn \l__clist_tmp_clist {#1}
19511   \exp_args:No \__clist_if_in_return:nnN \l__clist_tmp_clist {#2}
19512   \l__clist_tmp_clist
19513 }
19514 \cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
19515 {
19516   \__clist_if_wrap:nTF {#2}
19517   {
19518     \cs_set:Npe \__clist_tmp:w #1

```



```

19519         {
19520             \exp_not:N \tl_if_eq:nnT {##1}
19521             \exp_not:n
19522             {
19523                 {#2}
19524                 { \clist_map_break:n { \prg_return_true: \use_none:n } }
19525             }
19526         }
19527         \clist_map_function:NN #3 \__clist_tmp:w
19528         \prg_return_false:
19529     }
19530     {
19531         \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
19532         \tl_if_empty:oTF
19533         { \__clist_tmp:w ,#1, {} {} ,#2, }
19534         { \prg_return_false: } { \prg_return_true: }
19535     }
19536 }
19537 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
19538 { NV , No , Ne , c , cV , co , ce } { T , F , TF }
19539 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
19540 { nV , no , ne } { T , F , TF }

```

(End of definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `\__clist_if_in_return:nnN`. These functions are documented on page 194.)

66.7 Mapping over comma lists

`\clist_map_function:NN` If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing eight comma-delimited items at a time. The end is marked by `\s__clist_stop`, which may not appear in any of the items. Once the last group of eight items has been reached, we go through them more slowly using `\__clist_map_function_end:w`. The auxiliary function `\__clist_map_function:Nw` is also used in some other clist mappings.

```

19541 \cs_new:Npn \clist_map_function:NN #1#2
19542 {
19543     \clist_if_empty:NF #1
19544     {
19545         \exp_after:wN \__clist_map_function:Nw \exp_after:wN #2 #1 ,
19546         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19547         \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19548         \prg_break_point:Nn \clist_map_break: { }
19549     }
19550 }
19551 \cs_new:Npn \__clist_map_function:Nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
19552 {
19553     \__clist_use_none_delimit_by_s_stop:w
19554     #9 \__clist_map_function_end:w \s__clist_stop
19555     #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
19556     \__clist_map_function:Nw #1
19557 }
19558 \cs_new:Npn \__clist_map_function_end:w \s__clist_stop #1#2
19559 {

```

```

19560     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
19561     #1 {#2}
19562     \__clist_map_function_end:w \s__clist_stop
19563   }
19564 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End of definition for `\clist_map_function:NN`, `\__clist_map_function:Nw`, and `\__clist_map_function_end:w`. This function is documented on page 194.)

`\clist_map_function:nN` The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `\__clist_trim_next:w`. The auxiliary `\__clist_map_function_n:Nn` receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by `\__clist_map_unbrace:wn`.

```

19565 \cs_new:Npn \clist_map_function:nN #1#2
19566 {
19567   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
19568   \exp:w \__clist_trim_next:w \prg_do_nothing: #1 ,
19569   \s__clist_stop \clist_map_break: ,
19570   \prg_break_point:Nn \clist_map_break: { }
19571 }
19572 \cs_generate_variant:Nn \clist_map_function:nN { e }
19573 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
19574 {
19575   \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
19576   \__clist_map_unbrace:wn #2 , #1
19577   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
19578   \exp:w \__clist_trim_next:w \prg_do_nothing:
19579 }
19580 \cs_new:Npn \__clist_map_unbrace:wn #1, #2 { #2 {#1} }

```

(End of definition for `\clist_map_function:nN`, `\__clist_map_function_n:Nn`, and `\__clist_map_unbrace:wn`. This function is documented on page 194.)

`\clist_map_inline:Nn` `\clist_map_inline:cn` `\clist_map_inline:nn` Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with TeX’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

19581 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
19582 {
19583   \clist_if_empty:NF #1
19584   {
19585     \int_gincr:N \g__kernel_pr_g_map_int
19586     \cs_gset_protected:cpn
19587     { \__clist_map_ \int_use:N \g__kernel_pr_g_map_int :w } ##1 {#2}
19588     \exp_last_unbraced:Nco \__clist_map_function:Nw
19589     { \__clist_map_ \int_use:N \g__kernel_pr_g_map_int :w }
19590     #1 ,
19591     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19592     \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19593     \prg_break_point:Nn \clist_map_break:
19594     { \int_gdecr:N \g__kernel_pr_g_map_int }

```

```

19595     }
19596   }
19597   \cs_new_protected:Npn \clist_map_inline:nn #1
19598   {
19599     \clist_set:Nn \l__clist_tmp_clist {#1}
19600     \clist_map_inline:Nn \l__clist_tmp_clist
19601   }
19602   \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End of definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 195.)

`\clist_map_variable:NNn` The N-type version is a straightforward application of `\clist_map_tokens:Nn`, calling `\__clist_map_variable:Nnn` for each item to assign the variable and run the user's code. The n-type version is *not* implemented in terms of the n-type function `\clist_map_tokens:Nn`, because here we are allowed to clean up the n-type comma list non-expandably.

```

19603   \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
19604   { \clist_map_tokens:Nn #1 { \__clist_map_variable:Nnn #2 {#3} } }
19605   \cs_generate_variant:Nn \clist_map_variable:NNn { c }
19606   \cs_new_protected:Npn \__clist_map_variable:Nnn #1#2#3
19607   { \tl_set:Nn #1 {#3} #2 }
19608   \cs_new_protected:Npn \clist_map_variable:nNn #1
19609   {
19610     \clist_set:Nn \l__clist_tmp_clist {#1}
19611     \clist_map_variable:NNn \l__clist_tmp_clist
19612   }

```

(End of definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `\__clist_map_variable:Nnn`. These functions are documented on page 195.)

`\clist_map_tokens:Nn` Essentially a copy of `\clist_map_function:NN` with braces added.

```

\clist_map_tokens:cn
\__clist_map_tokens:nw
\__clist_map_tokens_end:w
19613   \cs_new:Npn \clist_map_tokens:Nn #1#2
19614   {
19615     \clist_if_empty:NF #1
19616     {
19617       \exp_last_unbraced:Nno \__clist_map_tokens:nw {#2} #1 ,
19618       \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19619       \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
19620       \prg_break_point:Nn \clist_map_break: { }
19621     }
19622   }
19623   \cs_new:Npn \__clist_map_tokens:nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
19624   {
19625     \__clist_use_none_delimit_by_s_stop:w
19626     #9 \__clist_map_tokens_end:w \s__clist_stop
19627     \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
19628     \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
19629     \__clist_map_tokens:nw {#1}
19630   }
19631   \cs_new:Npn \__clist_map_tokens_end:w \s__clist_stop \use:n #1#2
19632   {
19633     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
19634     #1 {#2}

```

```

19635     \__clist_map_tokens_end:w \s__clist_stop
19636   }
19637   \cs_generate_variant:Nn \clist_map_tokens:Nn { c }

```

(End of definition for `\clist_map_tokens:Nn`, `\__clist_map_tokens:nw`, and `\__clist_map_tokens_end:w`. This function is documented on page 195.)

`\clist_map_tokens:nn` Similar to `\clist_map_function:nN` but with a different way of grabbing items because `\__clist_map_tokens_n:nw` we cannot use `\exp_after:wN` to pass the `{code}`.

```

19638   \cs_new:Npn \clist_map_tokens:nn #1#2
19639   {
19640     \__clist_map_tokens_n:nw {#2}
19641     \prg_do_nothing: #1 , \s__clist_stop \clist_map_break: ,
19642     \prg_break_point:Nn \clist_map_break: { }
19643   }
19644   \cs_new:Npn \__clist_map_tokens_n:nw #1#2 ,
19645   {
19646     \tl_if_empty:oF { \use_none:nn #2 ? }
19647     {
19648       \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
19649       \tl_trim_spaces_apply:oN {#2} \use_ii_i:nn
19650       \__clist_map_unbrace:wn , {#1}
19651     }
19652     \__clist_map_tokens_n:nw {#1} \prg_do_nothing:
19653   }

```

(End of definition for `\clist_map_tokens:nn` and `\__clist_map_tokens_n:nw`. This function is documented on page 195.)

`\clist_map_break:` The break statements use the general `\prg_map_break:Nn` mechanism.
`\clist_map_break:n`

```

19654   \cs_new:Npn \clist_map_break:
19655   { \prg_map_break:Nn \clist_map_break: { } }
19656   \cs_new:Npn \clist_map_break:n
19657   { \prg_map_break:Nn \clist_map_break: }

```

(End of definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 195.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token
`\clist_count:c` count functions: turn each entry into a +1 then use integer evaluation to actually do the
`\clist_count:n` mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_`
`\clist_count:e` `function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop
`\__clist_count:n` manually, and skip blank items (but not `{}`, hence the extra spaces).
`\__clist_count:w`

```

19658   \cs_new:Npn \clist_count:N #1
19659   {
19660     \int_eval:n
19661     {
19662       0
19663       \clist_map_function:NN #1 \__clist_count:n
19664     }
19665   }
19666   \cs_generate_variant:Nn \clist_count:N { c }
19667   \cs_new:Npn \__clist_count:n #1 { + 1 }
19668   \cs_set_protected:Npn \__clist_tmp:w #1
19669   {

```

```

19670 \cs_new:Npn \clist_count:n ##1
19671 {
19672     \int_eval:n
19673     {
19674         0
19675         \__clist_count:w #1
19676         ##1 , \s__clist_stop \prg_break: , \prg_break_point:
19677     }
19678 }
19679 \cs_new:Npn \__clist_count:w ##1 ,
19680 {
19681     \__clist_use_none_delimit_by_s_stop:w ##1 \s__clist_stop
19682     \tl_if_blank:nF {##1} { + 1 }
19683     \__clist_count:w #1
19684 }
19685 }
19686 \exp_args:No \__clist_tmp:w \c_space_tl
19687 \cs_generate_variant:Nn \clist_count:n { e }

```

(End of definition for `\clist_count:N` and others. These functions are documented on page 196.)

66.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
\__clist_use:wwn
\__clist_use:nwwwnwn
\__clist_use:nwwn
\clist_use:Nn
\clist_use:cn

```

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

Otherwise, `\__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *<continuation>* function (`use_ii` or `use_iii` with its *<separator>* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\s__clist_stop`. The *<separator>* and the first of the three items are placed in the result, then we use the *<continuation>*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\s__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\s__clist_mark` is taken as a third item, and now the second `\s__clist_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

19688 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
19689 {
19690     \clist_if_exist:NTF #1
19691     {
19692         \int_case:nnF { \clist_count:N #1 }
19693         {
19694             { 0 } { }
19695             { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
19696             { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
19697         }
19698         {
19699             \exp_after:wN \__clist_use:nwwwnwn
19700             \exp_after:wN { \exp_after:wN } #1 ,
19701             \s__clist_mark , { \__clist_use:nwwwnwn {#3} }

```

```

19702         \s__clist_mark , { \__clist_use:nwwn {#4} }
19703         \s__clist_stop { }
19704     }
19705 }
19706 {
19707     \msg_expandable_error:nnn
19708     { kernel } { bad-variable } {#1}
19709 }
19710 }
19711 \cs_generate_variant:Nn \clist_use:Nnnn { c }
19712 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
19713 \cs_new:Npn \__clist_use:nwwwnwn
19714     #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
19715     { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
19716 \cs_new:Npn \__clist_use:nwwn #1#2 , #3 \s__clist_stop #4
19717     { \exp_not:n { #4 #1 #2 } }
19718 \cs_new:Npn \clist_use:Nn #1#2
19719     { \clist_use:Nnnn #1 {#2} {#2} {#2} }
19720 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End of definition for \clist_use:Nnnn and others. These functions are documented on page 196.)

\clist_use:N

\clist_use:c

```

19721 \cs_new_eq:NN \clist_use:N \tl_use:N
19722 \cs_generate_variant:Nn \clist_use:N { c }

```

(End of definition for \clist_use:N. This function is documented on page 197.)

\clist_use:nnnn

\clist_use:nn

__clist_use:Nw

__clist_use_one:w

__clist_use_end:w

__clist_use_more:w

Items are grabbed by __clist_use:Nw, which detects blank items with a \tl_if_empty:oTF test (in which case it recurses). Non-blank items are either the end of the list, in which case the argument #1 of __clist_use:Nw is used to properly end the list, or are normal items, which must be trimmed and properly unbraced. As we find successive items, the long list of __clist_use:Nw calls gets shortened and we end up calling __clist_use_more:w once we have found 3 items. This auxiliary leaves the first-found item and the general separator, and calls __clist_use:Nw to find more items. A subtlety is that we use __clist_use_end:w both in the case of a two-item list and for the last two items of a general list: to get the correct separator, __clist_use_more:w replaces the separator-of-two by the last-separator when called, namely as soon as we have found three items.

```

19723 \cs_new:Npn \clist_use:nnnn #1#2#3#4
19724     {
19725         \__clist_use:Nw \__clist_use_none_delimit_by_s_stop:w
19726         \__clist_use:Nw \__clist_use_one:w
19727         \__clist_use:Nw \__clist_use_end:w
19728         \__clist_use_more:w ;
19729         {#2} {#3} {#4} ;
19730         \prg_do_nothing: #1 , \s__clist_mark ,
19731         \s__clist_stop
19732     }
19733 \cs_new:Npn \__clist_use:Nw #1#2 ; #3 ; #4 ,
19734     {
19735         \tl_if_empty:oTF { \use_none:nn #4 ? }
19736         { \__clist_use:Nw #1#2 ; }
19737         {

```

```

19738     \_clist_use_none_delimit_by_s_mark:w #4 #1 \s__clist_mark
19739     \tl_trim_spaces_apply:oN {#4} \use_ii_i:nn
19740     \_clist_map_unbrace:wn , { #2 ; }
19741   }
19742   #3 ; \prg_do_nothing:
19743 }
19744 \cs_new:Npn \_clist_use_one:w \s__clist_mark #1 , #2#3#4 \s__clist_stop
19745 { \exp_not:n {#3} }
19746 \cs_new:Npn \_clist_use_end:w
19747   \s__clist_mark #1 , #2#3#4#5#6 \s__clist_stop
19748 { \exp_not:n { #4 #5 #3 } }
19749 \cs_new:Npn \_clist_use_more:w ; #1#2#3#4#5#6 ;
19750 {
19751   \exp_not:n { #3 #5 }
19752   \_clist_use:Nw \_clist_use_end:w \_clist_use_more:w ;
19753   {#1} {#2} {#6} {#5} {#6} ;
19754 }
19755 \cs_new:Npn \clist_use:nn #1#2 { \clist_use:nnnn {#1} {#2} {#2} {#2} }

```

(End of definition for `\clist_use:nnnn` and others. These functions are documented on page 197.)

66.9 Using a single item

<pre> \clist_item:Nn \clist_item:cn _clist_item:nnnN _clist_item:ffoN _clist_item:ffnN _clist_item_N_loop:nw </pre>	To avoid needing to test the end of the list at each step, we first compute the $\langle length \rangle$ of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat. <pre> 19756 \cs_new:Npn \clist_item:Nn #1#2 19757 { 19758 _clist_item:ffoN 19759 { \clist_count:N #1 } 19760 { \int_eval:n {#2} } 19761 #1 19762 _clist_item_N_loop:nw 19763 } 19764 \cs_new:Npn _clist_item:nnnN #1#2#3#4 19765 { 19766 \int_compare:nNnTF {#2} < 0 19767 { 19768 \int_compare:nNnTF {#2} < { - #1 } 19769 { _clist_use_none_delimit_by_s_stop:w } 19770 { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } } 19771 } 19772 { 19773 \int_compare:nNnTF {#2} > {#1} 19774 { _clist_use_none_delimit_by_s_stop:w } 19775 { #4 {#2} } 19776 } 19777 { } , #3 , \s__clist_stop 19778 } 19779 \cs_generate_variant:Nn _clist_item:nnnN { ffo, ff } 19780 \cs_new:Npn _clist_item_N_loop:nw #1 #2, </pre>

```

19781 {
19782   \int_compare:nNnTF {#1} = 0
19783     { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
19784     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
19785   }
19786 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End of definition for `\clist_item:Nn`, `\__clist_item:nnnN`, and `\__clist_item_N_loop:nw`. This function is documented on page 199.)

```

\clist_item:nn This starts in the same way as \clist_item:Nn by counting the items of the comma list.
\clist_item:Vn The final item should be space-trimmed before being brace-stripped, hence we insert a
\clist_item:vn couple of odd-looking \prg_do_nothing: to avoid losing braces. Blank items are ignored.
\clist_item:on
\clist_item:en
  \__clist_item_n:nw 19787 \cs_new:Npn \clist_item:nn #1#2
  \__clist_item_n_loop:nw 19788 {
  \__clist_item_n_end:n 19789   \__clist_item:ffnN
  \__clist_item_n_strip:n 19790     { \clist_count:n {#1} }
  \__clist_item_n_strip:w 19791     { \int_eval:n {#2} }
  \__clist_item_n_strip:w 19792     {#1}
  \__clist_item_n_strip:w 19793     \__clist_item_n:nw
  \__clist_item_n_strip:w 19794   }
  \cs_generate_variant:Nn \clist_item:nn { V , v , o , e }
  \cs_new:Npn \__clist_item_n:nw #1
  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
  \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
  {
  \exp_args:No \tl_if_blank:nTF {#2}
  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
  {
  \int_compare:nNnTF {#1} = 0
  { \exp_args:No \__clist_item_n_end:n {#2} }
  {
  \exp_args:Nf \__clist_item_n_loop:nw
  { \int_eval:n { #1 - 1 } }
  \prg_do_nothing:
  }
  }
  }
  \cs_new:Npn \__clist_item_n_end:n #1 #2 \s__clist_stop
  { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
  \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
  \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End of definition for `\clist_item:nn` and others. This function is documented on page 199.)

```

\clist_rand_item:n The N-type function is not implemented through the n-type function for efficiency: for
\clist_rand_item:N instance comma-list variables do not require space-trimming of their items. Even testing
\clist_rand_item:c for emptiness of an n-type comma-list is slow, so we count items first and use that both
\__clist_rand_item:nn for the emptiness test and the pseudo-random integer. Importantly, \clist_item:Nn
and \clist_item:nn only evaluate their argument once.
19816 \cs_new:Npn \clist_rand_item:n #1
19817 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
19818 \cs_new:Npn \__clist_rand_item:nn #1#2
19819 {

```



```

19820     \int_compare:nNnF {#1} = 0
19821     { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
19822   }
19823   \cs_new:Npn \clist_rand_item:N #1
19824   {
19825     \clist_if_empty:NF #1
19826     { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
19827   }
19828   \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End of definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `\__clist_rand_item:nn`. These functions are documented on page 199.)

66.10 Viewing comma lists

```

\clist_show:N Apply the general \__kernel_chk_tl_type:NnnT with \exp_not:o #2 serving as a
\clist_show:c dummy code to prevent a check performed by this auxiliary.
\clist_log:N
\clist_log:c
\__clist_show:NN
19829 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nneeee }
19830 \cs_generate_variant:Nn \clist_show:N { c }
19831 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nneeee }
19832 \cs_generate_variant:Nn \clist_log:N { c }
19833 \cs_new_protected:Npn \__clist_show:NN #1#2
19834 {
19835   \__kernel_chk_tl_type:NnnT #2 { clist } { \exp_not:o #2 }
19836   {
19837     \int_compare:nNnTF { \clist_count:N #2 }
19838     = { \exp_args:No \clist_count:n #2 }
19839     {
19840       #1 { clist } { show }
19841       { \token_to_str:N #2 }
19842       { \clist_map_function:NN #2 \msg_show_item:n }
19843       { } { }
19844     }
19845     {
19846       \msg_error:nnee { clist } { non-clist }
19847       { \token_to_str:N #2 } { \tl_to_str:N #2 }
19848     }
19849   }
19850 }

```

(End of definition for `\clist_show:N`, `\clist_log:N`, and `\__clist_show:NN`. These functions are documented on page 199.)

```

\clist_show:n A variant of the above: no existence check, empty first argument for the message.
\clist_log:n
\__clist_show:Nn
19851 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nneeee }
19852 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nneeee }
19853 \cs_new_protected:Npn \__clist_show:Nn #1#2
19854 {
19855   #1 { clist } { show }
19856   { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
19857 }

```

(End of definition for `\clist_show:n`, `\clist_log:n`, and `\__clist_show:Nn`. These functions are documented on page 199.)

66.11 Scratch comma lists

```
\l_tmpa_clist Temporary comma list variables.  
\l_tmpb_clist 19858 \clist_new:N \l_tmpa_clist  
\g_tmpa_clist 19859 \clist_new:N \l_tmpb_clist  
\g_tmpb_clist 19860 \clist_new:N \g_tmpa_clist  
19861 \clist_new:N \g_tmpb_clist
```

(End of definition for \l_tmpa_clist and others. These variables are documented on page 200.)

```
19862 \</code>
```

Chapter 67

l3token implementation

```
19863 <@@=char>
```

```
19864 <*code>
```

67.1 Internal auxiliaries

`\s__char_stop` Internal scan mark.

```
19865 \scan_new:N \s__char_stop
```

(End of definition for \s__char_stop.)

`\q__char_no_value` Internal recursion quarks.

```
19866 \quark_new:N \q__char_no_value
```

(End of definition for \q__char_no_value.)

`\__char_quark_if_no_value_p:N` Functions to query recursion quarks.

```
\__char_quark_if_no_value:NTF 19867 \__kernel_quark_new_conditional:Nn \__char_quark_if_no_value:N { TF }
```

(End of definition for __char_quark_if_no_value:NTF.)

67.2 Manipulating and interrogating character tokens

`\char_set_catcode:nn` Simple wrappers around the primitives.

`\char_value_catcode:n`

`\char_show_value_catcode:n`

```
19868 \cs_new_protected:Npn \char_set_catcode:nn #1#2
```

```
19869 { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
```

```
19870 \cs_new:Npn \char_value_catcode:n #1
```

```
19871 { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
```

```
19872 \cs_new_protected:Npn \char_show_value_catcode:n #1
```

```
19873 { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End of definition for \char_set_catcode:nn, \char_value_catcode:n, and \char_show_value_catcode:n. These functions are documented on page 204.)

```

\char_set_catcode_escape:N
  \char_set_catcode_group_begin:N
  \char_set_catcode_group_end:N
  \char_set_catcode_math_toggle:N
  \char_set_catcode_alignment:N
\char_set_catcode_end_line:N
  \char_set_catcode_parameter:N
  \char_set_catcode_math_superscript:N
  \char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

19874 \cs_new_protected:Npn \char_set_catcode_escape:N #1
19875   { \char_set_catcode:nn { '#1 } { 0 } }
19876 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
19877   { \char_set_catcode:nn { '#1 } { 1 } }
19878 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
19879   { \char_set_catcode:nn { '#1 } { 2 } }
19880 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
19881   { \char_set_catcode:nn { '#1 } { 3 } }
19882 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
19883   { \char_set_catcode:nn { '#1 } { 4 } }
19884 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
19885   { \char_set_catcode:nn { '#1 } { 5 } }
19886 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
19887   { \char_set_catcode:nn { '#1 } { 6 } }
19888 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
19889   { \char_set_catcode:nn { '#1 } { 7 } }
19890 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
19891   { \char_set_catcode:nn { '#1 } { 8 } }
19892 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
19893   { \char_set_catcode:nn { '#1 } { 9 } }
19894 \cs_new_protected:Npn \char_set_catcode_space:N #1
19895   { \char_set_catcode:nn { '#1 } { 10 } }
19896 \cs_new_protected:Npn \char_set_catcode_letter:N #1
19897   { \char_set_catcode:nn { '#1 } { 11 } }
19898 \cs_new_protected:Npn \char_set_catcode_other:N #1
19899   { \char_set_catcode:nn { '#1 } { 12 } }
19900 \cs_new_protected:Npn \char_set_catcode_active:N #1
19901   { \char_set_catcode:nn { '#1 } { 13 } }
19902 \cs_new_protected:Npn \char_set_catcode_comment:N #1
19903   { \char_set_catcode:nn { '#1 } { 14 } }
19904 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
19905   { \char_set_catcode:nn { '#1 } { 15 } }

```

(End of definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 203.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n
  \char_set_catcode_group_end:n
  \char_set_catcode_math_toggle:n
  \char_set_catcode_alignment:n
\char_set_catcode_end_line:n
  \char_set_catcode_parameter:n
  \char_set_catcode_math_superscript:n
  \char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

19906 \cs_new_protected:Npn \char_set_catcode_escape:n #1
19907   { \char_set_catcode:nn {#1} { 0 } }
19908 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
19909   { \char_set_catcode:nn {#1} { 1 } }
19910 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
19911   { \char_set_catcode:nn {#1} { 2 } }
19912 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
19913   { \char_set_catcode:nn {#1} { 3 } }
19914 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
19915   { \char_set_catcode:nn {#1} { 4 } }
19916 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
19917   { \char_set_catcode:nn {#1} { 5 } }
19918 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
19919   { \char_set_catcode:nn {#1} { 6 } }
19920 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
19921   { \char_set_catcode:nn {#1} { 7 } }

```

```

19922 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
19923 { \char_set_catcode:nn {#1} { 8 } }
19924 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
19925 { \char_set_catcode:nn {#1} { 9 } }
19926 \cs_new_protected:Npn \char_set_catcode_space:n #1
19927 { \char_set_catcode:nn {#1} { 10 } }
19928 \cs_new_protected:Npn \char_set_catcode_letter:n #1
19929 { \char_set_catcode:nn {#1} { 11 } }
19930 \cs_new_protected:Npn \char_set_catcode_other:n #1
19931 { \char_set_catcode:nn {#1} { 12 } }
19932 \cs_new_protected:Npn \char_set_catcode_active:n #1
19933 { \char_set_catcode:nn {#1} { 13 } }
19934 \cs_new_protected:Npn \char_set_catcode_comment:n #1
19935 { \char_set_catcode:nn {#1} { 14 } }
19936 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
19937 { \char_set_catcode:nn {#1} { 15 } }

```

(End of definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 203.)

```

\char_set_mathcode:nn Pretty repetitive, but necessary!
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n
19938 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
19939 { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19940 \cs_new:Npn \char_value_mathcode:n #1
19941 { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
19942 \cs_new_protected:Npn \char_show_value_mathcode:n #1
19943 { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
19944 \cs_new_protected:Npn \char_set_lccode:nn #1#2
19945 { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19946 \cs_new:Npn \char_value_lccode:n #1
19947 { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
19948 \cs_new_protected:Npn \char_show_value_lccode:n #1
19949 { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
19950 \cs_new_protected:Npn \char_set_uccode:nn #1#2
19951 { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19952 \cs_new:Npn \char_value_uccode:n #1
19953 { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
19954 \cs_new_protected:Npn \char_show_value_uccode:n #1
19955 { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
19956 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
19957 { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
19958 \cs_new:Npn \char_value_sfcode:n #1
19959 { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
19960 \cs_new_protected:Npn \char_show_value_sfcode:n #1
19961 { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End of definition for `\char_set_mathcode:nn` and others. These functions are documented on page 205.)

`\l_char_active_seq` Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

19962 \seq_new:N \l_char_special_seq
19963 \seq_set_split:Nnn \l_char_special_seq { }

```

```

19964 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
19965 \seq_new:N \l_char_active_seq
19966 \seq_set_split:Nnn \l_char_active_seq { }
19967 { \ " \$ & ^ _ \~ }

```

(End of definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 205.)

67.3 Creating character tokens

`\char_set_active_eq:NN` Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```

\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
19968 \group_begin:
19969   \char_set_catcode_active:N \^^@
19970   \cs_set_protected:Npn \__char_tmp:nN #1#2
19971   {
19972     \cs_new_protected:cpn { #1 :nN } ##1
19973     {
19974       \group_begin:
19975       \char_set_lccode:nn { '^^@ } { ##1 }
19976       \tex_lowercase:D { \group_end: #2 ^^@ }
19977     }
19978     \cs_new_protected:cpe { #1 :NN } ##1
19979     { \exp_not:c { #1 : nN } { '##1 } }
19980   }
19981   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
19982   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
19983 \group_end:
19984 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
19985 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
19986 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
19987 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End of definition for `\char_set_active_eq:NN` and others. These functions are documented on page 202.)

`\__char_int_to_roman:w` For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```

19988 \cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D

```

(End of definition for `\__char_int_to_roman:w`.)

`\__char_sep:`

```

19989 \cs_new_eq:NN \__char_sep: \__kernel_int_sep:

```

(End of definition for `\__char_sep:`.)

`\char_generate:nn` The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XeTeX, LuaTeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
  \l__char_tmp_tl
\__char_generate_invalid_catcode:
19990 \cs_new:Npn \char_generate:nn #1#2
19991 {
19992   \exp:w \exp_after:wN \__char_generate_aux:w

```

```

19993     \int_value:w \int_eval:n {#1} \exp_after:wN \__char_sep:
19994     \int_value:w \int_eval:n {#2} \__char_sep:
19995 }

```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTeX emulation only makes normal (charcode 32 spaces). However, `^~@` is filtered out separately as that can't be done with macro emulation either, so is treated separately. That done, hand off to the engine-dependent part.

```

19996 \cs_new:Npn \__char_generate_aux:w #1 \__char_sep: #2 \__char_sep:
19997 {
19998   \if_int_odd:w 0
19999     \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
20000     \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
20001     \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
20002     \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
20003     \msg_expandable_error:nn { char }
20004     { invalid-catcode }
20005   \else:
20006     \if_int_odd:w 0
20007       \if_int_compare:w #1 < \c_zero_int 1 \fi:
20008       \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
20009       \msg_expandable_error:nn { char }
20010       { out-of-range }
20011     \else:
20012       \if_int_compare:w #2#1 = 100 \exp_stop_f:
20013       \msg_expandable_error:nn { char } { null-space }
20014     \else:
20015       \__char_generate_aux:nnw {#1} {#2}
20016     \fi:
20017   \fi:
20018 \fi:
20019 \exp_end:
20020 }
20021 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. Recent (u)pTeX and the Unicode engines LuaTeX and XeTeX have engine-level support for expandable character creation. pdfTeX and older (u)pTeX releases do not. The branching here is low-level to avoid fixing the category code of the null character used in the false branch. The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTeX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

20022 \group_begin:
20023   \char_set_catcode_active:N \^^L
20024   \cs_set:Npn ^^L { }
20025   \if_cs_exist:N \tex_Ucharcat:D
20026     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
20027     {
20028       #3
20029       \exp_after:wN \exp_end:
20030       \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
20031     }
20032   \else:

```

For engines where `\Ucharcat` isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `\tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. The list is done in reverse as this puts the case of an active token *first*: that's needed to cover the possibility that it is `\outer`. Getting the braces into the list is done using some standard `\if_false:` manipulation, while all of the `\exp_not:N` are required as there is an expansion in the setup.

```

20033 \char_set_catcode_active:n { 0 }
20034 \tl_set:Nn \l__char_tmp_tl { \exp_not:N ^^@ \exp_not:N \or: }
20035 \char_set_catcode_other:n { 0 }
20036 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20037 \char_set_catcode_letter:n { 0 }
20038 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```

20039 \tl_put_right:Nn \l__char_tmp_tl { \use:n { ~ } \exp_not:N \or: }
20040 \tl_put_right:Nn \l__char_tmp_tl { \exp_not:N \or: }
20041 \char_set_catcode_math_subscript:n { 0 }
20042 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20043 \char_set_catcode_math_superscript:n { 0 }
20044 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20045 \char_set_catcode_parameter:n { 0 }
20046 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20047 \tl_put_right:Nn \l__char_tmp_tl { { \if_false: } \fi: \exp_not:N \or: }
20048 \char_set_catcode_alignment:n { 0 }
20049 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20050 \char_set_catcode_math_toggle:n { 0 }
20051 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
20052 \char_set_catcode_group_end:n { 0 }
20053 \tl_put_right:Nn \l__char_tmp_tl { \if_false: { \fi: ^^@ \exp_not:N \or: } % }
20054 \char_set_catcode_group_begin:n { 0 } % {
20055 \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: } }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The e-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list.

```

20056 \cs_set_protected:Npn \__char_tmp:n #1
20057 {
20058   \char_set_lccode:nn { 0 } {#1}
20059   \char_set_lccode:nn { 32 } {#1}
20060   \exp_args:Ne \tex_lowercase:D
20061   {
20062     \tl_const:Ne
20063     \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
20064     { \exp_not:o \l__char_tmp_tl }
20065   }
20066 }
20067 \int_step_function:nnN { 0 } { 255 } \__char_tmp:n

```

As $\text{\TeX}$ is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. $\text{\TeX}$ is happy if the token is hidden between braces within `\if_false: ... \fi:`. The rather low-level approach here

expands in one step to the $\langle target\ token\rangle$ ($\backslash or: \dots$), then $\backslash exp\_after:wN\ \langle target\ token\rangle$ ($\backslash or: \dots$) expands in one step to $\langle target\ token\rangle$. This means that $\backslash exp\_not:N$ is applied to a potentially-problematic active token.

```

20068 \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
20069 {
20070     #3
20071     \if_false: { \fi:
20072         \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
20073         \exp_after:wN \exp_after:wN
20074         \if_case:w \tex_numexpr:D 13 - #2
20075             \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
20076             \exp_after:wN \exp_after:wN \exp_after:wN \scan_stop:
20077             \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N
20078             \cs:w c__char_ \__char_int_to_roman:w #1 _tl \cs_end:
20079         }
20080     \fi:
20081 }
20082 \fi:
20083 \group_end:

```

(End of definition for $\backslash char\_generate:nn$ and others. This function is documented on page 202.)

$\backslash c\_catcode\_active\_space\_tl$ While $\backslash char\_generate:nn$ can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive $\backslash tex\_lowercase:D$ trick.

```

20084 \group_begin:
20085     \char_set_catcode_active:N *
20086     \char_set_lccode:nn { '*' } { '\ }
20087     \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
20088 \group_end:

```

(End of definition for $\backslash c\_catcode\_active\_space\_tl$. This variable is documented on page 202.)

$\backslash c\_catcode\_other\_space\_tl$ Create a space with category code 12: an “other” space.

```

20089 \tl_const:Nc \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }

```

(End of definition for $\backslash c\_catcode\_other\_space\_tl$. This function is documented on page 203.)

67.4 Generic tokens

```

20090 \<@@=token>

```

$\backslash s\_token\_mark$ Internal scan marks.

```

\backslash s\_token\_stop
20091 \scan_new:N \s_token_mark
20092 \scan_new:N \s_token_stop

```

(End of definition for $\backslash s\_token\_mark$ and $\backslash s\_token\_stop$.)

$\backslash token\_to\_meaning:N$ These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

$\backslash token\_to\_meaning:c$

$\backslash token\_to\_str:N$

$\backslash token\_to\_str:c$

(End of definition for $\backslash token\_to\_meaning:N$ and $\backslash token\_to\_str:N$. These functions are documented on page 206.)

`\token_to_catcode:N` The macro works by comparing the input token with `\if_catcode:w` with all valid category codes. Since the most common tokens in an average argument list are of category 11 or 12 those are tested first. And since a space and braces are no ordinary N-type arguments, and only control sequences let to those categories can match them they are tested last.

```

20093 \cs_new:Npn \token_to_catcode:N
20094 { \int_value:w \group_align_safe_begin: \__token_to_catcode:N }
20095 \cs_new:Npn \__token_to_catcode:N #1
20096 {
20097   \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
20098     11
20099   \else:
20100     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
20101       12
20102     \else:
20103       \if_catcode:w \exp_not:N #1 \c_math_toggle_token
20104         3
20105       \else:
20106         \if_catcode:w \exp_not:N #1 \c_alignment_token
20107           4
20108         \else:
20109           \if_catcode:w \exp_not:N #1 ##
20110             6
20111           \else:
20112             \if_catcode:w \exp_not:N #1 \c_math_superscript_token
20113               7
20114             \else:
20115               \if_catcode:w \exp_not:N #1 \c_math_subscript_token
20116                 8
20117             \else:
20118               \if_catcode:w \exp_not:N #1 \c_group_begin_token
20119                 1
20120             \else:
20121               \if_catcode:w \exp_not:N #1 \c_group_end_token
20122                 2
20123             \else:
20124               \if_catcode:w \exp_not:N #1 \c_space_token
20125                 10
20126             \else:
20127               \token_if_cs:NTF #1 { 16 } { 13 }
20128               \fi:
20129             \fi:
20130             \fi:
20131             \fi:
20132             \fi:
20133             \fi:
20134             \fi:
20135             \fi:
20136             \fi:
20137             \fi:
20138           \group_align_safe_end:
20139           \exp_stop_f:
20140 }

```

(End of definition for `\token_to_catcode:N` and `\__token_to_catcode:N`. This function is documented on page 206.)

```

\c_group_begin_token We define these useful tokens. For the brace and space tokens things have to be done
\c_group_end_token by hand: the formal argument spec. for \cs_new_eq:NN does not cover them so we do
\c_math_toggle_token things by hand. (As currently coded it would work with \cs_new_eq:NN but that's not
\c_alignment_token really a great idea to show off: we want people to stick to the defined interfaces and that
\c_parameter_token includes us.) So that these few odd names go into the log when appropriate there is a
\c_math_superscript_token need to hand-apply the \__kernel_chk_if_free_cs:N check.
\c_math_subscript_token
\c_space_token
\c_catcode_letter_token
\c_catcode_other_token
20141 \group_begin:
20142 \__kernel_chk_if_free_cs:N \c_group_begin_token
20143 \tex_global:D \tex_let:D \c_group_begin_token {
20144 \__kernel_chk_if_free_cs:N \c_group_end_token
20145 \tex_global:D \tex_let:D \c_group_end_token }
20146 \char_set_catcode_math_toggle:N \*
20147 \cs_new_eq:NN \c_math_toggle_token *
20148 \char_set_catcode_alignment:N \*
20149 \cs_new_eq:NN \c_alignment_token *
20150 \cs_new_eq:NN \c_parameter_token #
20151 \cs_new_eq:NN \c_math_superscript_token ^
20152 \char_set_catcode_math_subscript:N \*
20153 \cs_new_eq:NN \c_math_subscript_token *
20154 \__kernel_chk_if_free_cs:N \c_space_token
20155 \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
20156 \cs_new_eq:NN \c_catcode_letter_token a
20157 \cs_new_eq:NN \c_catcode_other_token 1
20158 \group_end:

```

(End of definition for `\c_group_begin_token` and others. These functions are documented on page 206.)

```

\__token_active_tl Not an implicit token!
20159 \group_begin:
20160 \char_set_catcode_active:N \*
20161 \tl_const:Nn \__token_active_tl { \exp_not:N * }
20162 \group_end:

```

(End of definition for `\__token_active_tl`.)

67.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

`\token_if_group_begin:NTF`

```

20163 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
20164 {
20165 \if_catcode:w \exp_not:N #1 \c_group_begin_token
20166 \prg_return_true: \else: \prg_return_false: \fi:
20167 }

```

(End of definition for `\token_if_group_begin:N`. This function is documented on page 207.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

`\token_if_group_end:NTF`

```

20168 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
20169 {

```

```

20170     \if_catcode:w \exp_not:N #1 \c_group_end_token
20171     \prg_return_true: \else: \prg_return_false: \fi:
20172   }

```

(End of definition for `\token_if_group_end:NTF`. This function is documented on page 207.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

`\token_if_math_toggle:NTF`

```

20173 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
20174 {
20175     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
20176     \prg_return_true: \else: \prg_return_false: \fi:
20177 }

```

(End of definition for `\token_if_math_toggle:NTF`. This function is documented on page 207.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

`\token_if_alignment:NTF`

```

20178 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
20179 {
20180     \if_catcode:w \exp_not:N #1 \c_alignment_token
20181     \prg_return_true: \else: \prg_return_false: \fi:
20182 }

```

(End of definition for `\token_if_alignment:NTF`. This function is documented on page 207.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.

`\token_if_parameter:NTF`

We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```

20183 \group_begin:
20184 \cs_set_eq:NN \c_parameter_token \scan_stop:
20185 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
20186 {
20187     \if_catcode:w \exp_not:N #1 \c_parameter_token
20188     \prg_return_true: \else: \prg_return_false: \fi:
20189 }
20190 \group_end:

```

(End of definition for `\token_if_parameter:NTF`. This function is documented on page 207.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

`\token_if_math_superscript:NTF`

```

20191 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
20192 { p , T , F , TF }
20193 {
20194     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
20195     \prg_return_true: \else: \prg_return_false: \fi:
20196 }

```

(End of definition for `\token_if_math_superscript:NTF`. This function is documented on page 207.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.
`\token_if_math_subscript:NTF`

```
20197 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
20198 {
20199     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
20200     \prg_return_true: \else: \prg_return_false: \fi:
20201 }
```

(End of definition for \token_if_math_subscript:NTF. This function is documented on page 207.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

`\token_if_space:NTF`

```
20202 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
20203 {
20204     \if_catcode:w \exp_not:N #1 \c_space_token
20205     \prg_return_true: \else: \prg_return_false: \fi:
20206 }
```

(End of definition for \token_if_space:NTF. This function is documented on page 207.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

`\token_if_letter:NTF`

```
20207 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
20208 {
20209     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
20210     \prg_return_true: \else: \prg_return_false: \fi:
20211 }
```

(End of definition for \token_if_letter:NTF. This function is documented on page 208.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.
`\token_if_other:NTF`

```
20212 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
20213 {
20214     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
20215     \prg_return_true: \else: \prg_return_false: \fi:
20216 }
```

(End of definition for \token_if_other:NTF. This function is documented on page 208.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_token_active_tl` for this. A technical point is that `\c__token_active_tl` is in fact a macro expanding to `\exp_not:N *`, where `*` is active.

`\token_if_active:NTF`

```
20217 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
20218 {
20219     \if_catcode:w \exp_not:N #1 \c__token_active_tl
20220     \prg_return_true: \else: \prg_return_false: \fi:
20221 }
```

(End of definition for \token_if_active:NTF. This function is documented on page 208.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NNTF`

```
20222 \prg_new_eq_conditional:NNn \token_if_eq_meaning:NN \cs_if_eq:NN
20223 { p , T , F , TF }
```

(End of definition for \token_if_eq_meaning:NNTF. This function is documented on page 208.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```

20224 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
20225 {
20226   \if_catcode:w \exp_not:N #1 \exp_not:N #2
20227   \prg_return_true: \else: \prg_return_false: \fi:
20228 }

```

(End of definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 208.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```

20229 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
20230 {
20231   \if_charcode:w \exp_not:N #1 \exp_not:N #2
20232   \prg_return_true: \else: \prg_return_false: \fi:
20233 }

```

(End of definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 208.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like
`\token_if_macro:NNTF` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`\__token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form
`\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyze what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:.` We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```

20234 \use:e
20235 {
20236   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
20237   { p , T , F , TF }
20238   {
20239     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
20240     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma : }
20241     \s__token_stop
20242   }
20243   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
20244   #1 \tl_to_str:n { ma } #2 \c_colon_str #3 \s__token_stop
20245   }
20246   {
20247     \str_if_eq:nnTF { #2 } { cro }
20248     { \prg_return_true: }
20249     { \prg_return_false: }
20250   }

```

(End of definition for `\token_if_macro:NNTF` and `\__token_if_macro_p:w`. This function is documented on page 208.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

20251 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
20252 {
20253   \if_catcode:w \exp_not:N #1 \scan_stop:
20254   \prg_return_true: \else: \prg_return_false: \fi:
20255 }

```

(End of definition for `\token_if_cs:NTF`. This function is documented on page 208.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX temporarily converts `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of T_EX’s conditional apparatus).

`\token_if_expandable:NTF`

```

20256 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
20257 {
20258   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
20259   \prg_return_false:
20260   \else:
20261     \if_cs_exist:N #1
20262     \prg_return_true:
20263     \else:
20264     \prg_return_false:
20265     \fi:
20266   \fi:
20267 }

```

(End of definition for `\token_if_expandable:NTF`. This function is documented on page 208.)

`\__token_delimit_by_char:w` These auxiliary functions are used below to define some conditionals which detect whether
`\__token_delimit_by_count:w` the `\meaning` of their argument begins with a particular string. Each auxiliary takes an
`\__token_delimit_by_dimen:w` argument delimited by a string, a second one delimited by `\s__token_stop`, and returns
`\__token_delimit_by_font:w` the first one and its delimiter. This result is eventually compared to another string. Note
`\__token_delimit_by_macro:w` that the “font” auxiliary is delimited by a space followed by “font”. This avoids an
`\__token_delimit_by_muskip:w` unnecessary check for the `\font` primitive below.
`\__token_delimit_by_skip:w`
`\__token_delimit_by_toks:w`

```

20268 \group_begin:
20269 \cs_set_protected:Npn \__token_tmp:w #1
20270 {
20271   \use:e
20272   {
20273     \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
20274     ##1 \tl_to_str:n {#1} ##2 \s__token_stop
20275     { ##1 \tl_to_str:n {#1} }
20276   }
20277 }
20278 \__token_tmp:w { char" }
20279 \__token_tmp:w { count }
20280 \__token_tmp:w { dimen }
20281 \__token_tmp:w { ~ font }
20282 \__token_tmp:w { macro }
20283 \__token_tmp:w { muskip }
20284 \__token_tmp:w { skip }
20285 \__token_tmp:w { toks }
20286 \group_end:

```

(End of definition for `\__token_delimit_by_char:w` and others.)

Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within `e`-expansion. The temporary function `\__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be `false` (thanks to the leading space for `font`), and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two `TeX` primitives which would wrongly be recognized as registers otherwise. Despite using `TeX`'s primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not `TeX` conditionals).

```

20287 \group_begin:
20288 \cs_set_protected:Npn \__token_tmp:w #1#2#3
20289 {
20290   \use:e
20291   {
20292     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ##1
20293     { p , T , F , TF }
20294     {
20295       \cs_if_exist:cT { tex_ #2 :D }
20296       {
20297         \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 :D }
20298         \exp_not:N \prg_return_false:
20299         \exp_not:N \else:
20300         \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 def:D }
20301         \exp_not:N \prg_return_false:
20302         \exp_not:N \else:
20303       }
20304       \exp_not:N \str_if_eq:eeTF
20305       {
20306         \exp_not:N \exp_after:wN
20307         \exp_not:c { __token_delimit_by_ #2 :w }
20308         \exp_not:N \token_to_meaning:N ##1
20309         ? \tl_to_str:n {#2} \s__token_stop

```



```

20310         }
20311         { \exp_not:n {#3} }
20312         { \exp_not:N \prg_return_true: }
20313         { \exp_not:N \prg_return_false: }
20314         \cs_if_exist:cT { tex_ #2 :D }
20315         {
20316             \exp_not:N \fi:
20317             \exp_not:N \fi:
20318         }
20319     }
20320 }
20321 }
20322 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
20323 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
20324 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
20325 \__token_tmp:w { protected_macro } { macro }
20326     { \tl_to_str:n { \protected } macro }
20327 \__token_tmp:w { protected_long_macro } { macro }
20328     { \token_to_str:N \protected \tl_to_str:n { \long } macro }
20329 \__token_tmp:w { font_selection } { ~ font } { select ~ font }
20330 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
20331 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
20332 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
20333 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
20334 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
20335 \group_end:

```

(End of definition for `\token_if_chardef:NTF` and others. These functions are documented on page 209.)

`\token_if_control_word_p:N`
`\token_if_control_word:N $\mathbf{T}\mathbf{F}$`
`\__token_if_control_word_aux:w`

This checks that the token consists of the escape character followed by one or more letters. As #1 could be a control symbol let to `\else:` or `\fi:`, etc., so we use an auxiliary function to grab the second `\if:w`.

```

20336 \prg_new_conditional:Npnn \token_if_control_word:N #1 { p , T , F , TF }
20337 {
20338     \if:w \exp_not:N #1 \token_to_str:N #1
20339         \__token_if_control_word_false:w
20340     \fi:
20341     \if:w 0 \tex_strcmp:D { \exp_not:N #1 } { \token_to_str:N #1 }
20342         \__token_if_control_word_false:w
20343     \fi:
20344     \if_true:
20345         \prg_return_true:
20346     \else:
20347         \prg_return_false:
20348     \fi:
20349 }
20350 \cs_new:Npn \__token_if_control_word_false:w \fi: #1 \if_true: { \fi: \if_false: }

```

(End of definition for `\token_if_control_word:N $\mathbf{T}\mathbf{F}$` and `\__token_if_control_word_aux:w`. This function is documented on page 209.)

`\token_if_control_symbol_p:N`
`\token_if_control_symbol:N $\mathbf{T}\mathbf{F}$`
`\__token_if_control_symbol_false:w`

A similar plan but we need to test for the space behavior, and then whether we are dealing with a control sequence. In this case there is no worry about a dangling token.

```

20351 \prg_new_conditional:Npnn \token_if_control_symbol:N #1 { p , T , F , TF }
20352 {
20353   \if_catcode:w \exp_not:N #1 \scan_stop:
20354   \else:
20355     \__token_if_control_symbol_false:w
20356   \fi:
20357   \if:w 0 \tex_strcmp:D { \exp_not:N #1 } { \token_to_str:N #1 }
20358   \prg_return_true:
20359   \else:
20360   \prg_return_false:
20361   \fi:
20362 }
20363 \cs_new:Npn \__token_if_control_symbol_false:w \fi: \if:w 0 \tex_strcmp:D #1#2
20364 { \fi: \if_false: }

```

(End of definition for `\token_if_control_symbol:NTF` and `\__token_if_control_symbol_false:w`. This function is documented on page 209.)

`\token_if_primitive_p:N`

`\token_if_primitive:NTF`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is undefined, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form `<letters>:<user material>`. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than ‘A’ (this is not quite a test for “only letters”, but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\tex_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

For LuaTeX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

In LuaMetaTeX some of the command names are different, so we check for both versions. The first one is always the LuaTeX version.

```

20365 \sys_if_engine luatex:TF
20366 {

```

```

20367 </code>
20368 (*lua)
20369 do
20370     local get_command = token.get_command
20371     local get_index = token.get_index
20372     local get_mode = token.get_mode or token.get_index
20373     local cmd = command_id
20374     local set_font = cmd'get_font'
20375     local biggest_char = token.biggest_char and token.biggest_char()
20376                     or status.getconstants().max_character_code
20377
20378     local mode_below_biggest_char = {}
20379     local index_not_nil = {}
20380     local mode_not_null = {}
20381     local non_primitive = {
20382         [cmd'left_brace'] = true,
20383         [cmd'right_brace'] = true,
20384         [cmd'math_shift'] = true,
20385         [cmd'mac_param' or cmd'parameter'] = mode_below_biggest_char,
20386         [cmd'sup_mark' or cmd'superscript'] = true,
20387         [cmd'sub_mark' or cmd'subscript'] = true,
20388         [cmd'endv' or cmd'ignore'] = true,
20389         [cmd'spacer'] = true,
20390         [cmd'letter'] = true,
20391         [cmd'other_char'] = true,
20392         [cmd'tab_mark' or cmd'alignment_tab'] = mode_below_biggest_char,
20393         [cmd'char_given'] = true,
20394         [cmd'math_given' or 'math_char_given'] = true,
20395         [cmd'xmath_given' or 'math_char_xgiven'] = true,
20396         [cmd'set_font'] = mode_not_null,
20397         [cmd'undefined_cs'] = true,
20398         [cmd'call'] = true,
20399         [cmd'long_call' or cmd'protected_call'] = true,
20400         [cmd'outer_call' or cmd'tolerant_call'] = true,
20401         [cmd'long_outer_call' or cmd'tolerant_protected_call'] = true,
20402         [cmd'assign_glue' or cmd'register_glue'] = index_not_nil,
20403         [cmd'assign_mu_glue' or cmd'register_mu_glue' or cmd'register_muglue'] = index_not_nil,
20404         [cmd'assign_toks' or cmd'register_toks'] = index_not_nil,
20405         [cmd'assign_int' or cmd'register_int' or cmd'register_integer'] = index_not_nil,
20406         [cmd'assign_attr' or cmd'register_attribute'] = true,
20407         [cmd'assign_dimen' or cmd'register_dimen' or cmd'register_dimension'] = index_not_nil,
20408     }
20409
20410     luacmd("__token_if_primitive_lua:N", function()
20411         local tok = get_next()
20412         local is_non_primitive = non_primitive[get_command(tok)]
20413         return put_next(
20414             is_non_primitive == true
20415             and false_tok
20416         or is_non_primitive == nil
20417             and true_tok
20418         or is_non_primitive == mode_not_null
20419             and (get_mode(tok) == 0 and true_tok or false_tok)
20420         or is_non_primitive == index_not_nil

```

```

20421         and (get_index(tok) and false_tok or true_tok)
20422     or is_non_primitive == mode_below_biggest_char
20423     and (get_mode(tok) > biggest_char and true_tok or false_tok))
20424 end, "global")
20425 end
20426 \lua
20427 (*code)
20428 \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
20429 {
20430     \__token_if_primitive_lua:N #1
20431 }
20432 }
20433 {
20434     \tex_global:D \tex_chardef:D \c__token_A_int = 'A ~ %
20435     \use:e
20436     {
20437         \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N #1
20438         { p , T , F , TF }
20439         {
20440             \exp_not:N \token_if_macro:NTF #1
20441             \exp_not:N \prg_return_false:
20442             {
20443                 \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
20444                 \exp_not:N \token_to_meaning:N #1
20445                 \tl_to_str:n { : : : } \s__token_stop #1
20446             }
20447         }
20448         \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
20449         #1#2 #3 \c_colon_str #4 \s__token_stop
20450         {
20451             \exp_not:N \tl_if_empty:oTF
20452             { \exp_not:N \__token_if_primitive_space:w #3 ~ }
20453             {
20454                 \exp_not:N \__token_if_primitive_loop:N #3
20455                 \c_colon_str \s__token_stop
20456             }
20457             { \exp_not:N \__token_if_primitive_nullfont:N }
20458         }
20459     }
20460     \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
20461     \cs_new:Npn \__token_if_primitive_nullfont:N #1
20462     {
20463         \if_meaning:w \tex_nullfont:D #1
20464         \prg_return_true:
20465     \else:
20466         \prg_return_false:
20467     \fi:
20468 }
20469 \cs_new:Npn \__token_if_primitive_loop:N #1
20470 {
20471     \if_int_compare:w '#1 < \c__token_A_int %
20472     \exp_after:wN \__token_if_primitive:Nw
20473     \exp_after:wN #1
20474 \else:

```

```

20475         \exp_after:wN \__token_if_primitive_loop:N
20476     \fi:
20477 }
20478 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \s__token_stop
20479 {
20480     \if:w : #1
20481         \exp_after:wN \__token_if_primitive_undefined:N
20482     \else:
20483         \prg_return_false:
20484         \exp_after:wN \use_none:n
20485     \fi:
20486 }
20487 \cs_new:Npn \__token_if_primitive_undefined:N #1
20488 {
20489     \if_cs_exist:N #1
20490         \prg_return_true:
20491     \else:
20492         \prg_return_false:
20493     \fi:
20494 }
20495 }

```

(End of definition for \token_if_primitive:NTF and others. This function is documented on page 210.)

```

\token_case_catcode:Nn
\token_case_catcode:NnTF
\token_case_charcode:Nn
\token_case_charcode:NnTF
\token_case_meaning:Nn
\token_case_meaning:NnTF
__token_case:NNnTF
__token_case:NNw
__token_case_end:nw

```

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the true or false branch code is inserted.

```

20496 \cs_new:Npn \token_case_catcode:Nn #1#2
20497 { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } { } }
20498 \cs_new:Npn \token_case_catcode:NnT #1#2#3
20499 { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} {#3} { } }
20500 \cs_new:Npn \token_case_catcode:NnF #1#2
20501 { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } }
20502 \cs_new:Npn \token_case_catcode:NnTF
20503 { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF }
20504 \cs_new:Npn \token_case_charcode:Nn #1#2
20505 { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } { } }
20506 \cs_new:Npn \token_case_charcode:NnT #1#2#3
20507 { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} {#3} { } }
20508 \cs_new:Npn \token_case_charcode:NnF #1#2
20509 { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } }
20510 \cs_new:Npn \token_case_charcode:NnTF
20511 { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF }
20512 \cs_new:Npn \token_case_meaning:Nn #1#2
20513 { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } { } }
20514 \cs_new:Npn \token_case_meaning:NnT #1#2#3
20515 { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} {#3} { } }
20516 \cs_new:Npn \token_case_meaning:NnF #1#2
20517 { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } }
20518 \cs_new:Npn \token_case_meaning:NnTF
20519 { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF }
20520 \cs_new:Npn \__token_case:NNnTF #1#2#3#4#5

```

```

20521 {
20522   \__token_case:NNw #1 #2 #3 #2 { }
20523   \s__token_mark {#4}
20524   \s__token_mark {#5}
20525   \s__token_stop
20526 }
20527 \cs_new:Npn \__token_case:NNw #1#2#3#4
20528 {
20529   #1 #2 #3
20530   { \__token_case_end:nw {#4} }
20531   { \__token_case:NNw #1 #2 }
20532 }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare \s__token_mark and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \s__token_mark and so #4 is the **false** code (the **true** code is mopped up by #3).

```

20533 \cs_new:Npn \__token_case_end:nw #1#2#3 \s__token_mark #4#5 \s__token_stop
20534 { \exp_end: #1 #4 }

```

(End of definition for \token_case_catcode:NnTF and others. These functions are documented on page 211.)

67.6 Peeking ahead at the next token

```

20535 <@@=peek>

```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token Storage tokens which are publicly documented: the token peeked.

```

\g_peek_token
20536 \cs_new_eq:NN \l_peek_token ?
20537 \cs_new_eq:NN \g_peek_token ?

```

(End of definition for \l_peek_token and \g_peek_token. These variables are documented on page 211.)

\l__peek_search_token The token to search for as an implicit token: cf. \l__peek_search_tl.

```

20538 \cs_new_eq:NN \l__peek_search_token ?

```

(End of definition for \l__peek_search_token.)

\l__peek_search_tl The token to search for as an explicit token: cf. \l__peek_search_token.

```

20539 \tl_new:N \l__peek_search_tl

```

(End of definition for \l_peek_search_tl.)

_peek_true:w Functions used by the branching and space-stripping code.
_peek_true_aux:w 20540 \cs_new:Npn _peek_true:w { }
_peek_false:w 20541 \cs_new:Npn _peek_true_aux:w { }
_peek_tmp:w 20542 \cs_new:Npn _peek_false:w { }
20543 \cs_new:Npn _peek_tmp:w { }

(End of definition for _peek_true:w and others.)

\s_peek_mark Internal scan marks.

\s_peek_stop 20544 \scan_new:N \s_peek_mark
20545 \scan_new:N \s_peek_stop

(End of definition for \s_peek_mark and \s_peek_stop.)

_peek_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.

20546 \cs_new:Npn _peek_use_none_delimit_by_s_stop:w #1 \s_peek_stop { }

(End of definition for _peek_use_none_delimit_by_s_stop:w.)

\peek_after:Nw Simple wrappers for \futurelet: no arguments absorbed here.

\peek_gafter:Nw 20547 \cs_new_protected:Npn \peek_after:Nw
20548 { \tex_futurelet:D \l_peek_token }
20549 \cs_new_protected:Npn \peek_gafter:Nw
20550 { \tex_global:D \tex_futurelet:D \g_peek_token }

(End of definition for \peek_after:Nw and \peek_gafter:Nw. These functions are documented on page 211.)

_peek_true_remove:w A function to remove the next token and then regain control.

20551 \cs_new_protected:Npn _peek_true_remove:w
20552 {
20553 \tex_afterassignment:D _peek_true_aux:w
20554 \cs_set_eq:NN _peek_tmp:w
20555 }

(End of definition for _peek_true_remove:w.)

\peek_remove_spaces:n Repeatedly use _peek_true_remove:w to remove a space and call _peek_true_remove_spaces:w.

_peek_remove_spaces: 20556 \cs_new_protected:Npn \peek_remove_spaces:n #1
20557 {
20558 \cs_set:Npe _peek_false:w { \exp_not:n {#1} }
20559 \group_align_safe_begin:
20560 \cs_set:Npn _peek_true_aux:w { \peek_after:Nw _peek_remove_spaces: }
20561 _peek_true_aux:w
20562 }
20563 \cs_new_protected:Npn _peek_remove_spaces:
20564 {
20565 \if_meaning:w \l_peek_token \c_space_token
20566 \exp_after:wN _peek_true_remove:w
20567 \else:
20568 \group_align_safe_end:
20569 \exp_after:wN _peek_false:w
20570 \fi:
20571 }

(End of definition for `\peek_remove_spaces:n` and `\__peek_remove_spaces:.` This function is documented on page 212.)

`\peek_remove_filler:n` Here we expand the input, removing spaces and `\scan_stop:` tokens until we reach a non-expandable token. At that stage we re-insert the payload. To deal with the problem of `&` tokens, we have to put the align-safe group in the correct place.

`\__peek_remove_filler:w`

`\__peek_remove_filler:`

`\__peek_remove_filler_expand:w`

```

20572 \cs_new_protected:Npn \peek_remove_filler:n #1
20573 {
20574   \cs_set:Npn \__peek_true_aux:w { \__peek_remove_filler:w }
20575   \cs_set:Npe \__peek_false:w
20576   {
20577     \exp_not:N \group_align_safe_end:
20578     \exp_not:n {#1}
20579   }
20580   \group_align_safe_begin:
20581   \__peek_remove_filler:w
20582 }
20583 \cs_new_protected:Npn \__peek_remove_filler:w
20584 {
20585   \exp_after:wN \peek_after:Nw \exp_after:wN \__peek_remove_filler:
20586   \exp:w \exp_end_continue_f:w
20587 }

```

Here we can nest conditionals as `\l_peek_token` is only skipped over in the nested one if it's a space: no problems with conditionals or outer tokens.

```

20588 \cs_new_protected:Npn \__peek_remove_filler:
20589 {
20590   \if_catcode:w \exp_not:N \l_peek_token \c_space_token
20591     \exp_after:wN \__peek_true_remove:w
20592   \else:
20593     \if_meaning:w \l_peek_token \scan_stop:
20594       \exp_after:wN \exp_after:wN \exp_after:wN
20595       \__peek_true_remove:w
20596     \else:
20597       \exp_after:wN \exp_after:wN \exp_after:wN
20598       \__peek_remove_filler_expand:w
20599     \fi:
20600   \fi:
20601 }

```

To deal with undefined control sequences in the same way TeX does, we need to check for expansion manually.

```

20602 \cs_new_protected:Npn \__peek_remove_filler_expand:w
20603 {
20604   \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
20605   \exp_after:wN \__peek_false:w
20606   \else:
20607     \exp_after:wN \__peek_remove_filler:w
20608   \fi:
20609 }

```

(End of definition for `\peek_remove_filler:n` and others. This function is documented on page 213.)

`\__peek_token_generic_aux:NNTF` The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed

here as the input stream needs to be cleared for the peek function itself. Here, #1 is `\__peek_true_remove:w` when removing the token and `\__peek_true_aux:w` otherwise.

```

20610 \cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
20611 {
20612   \group_align_safe_begin:
20613   \cs_set_eq:NN \l__peek_search_token #3
20614   \tl_set:Nn \l__peek_search_tl {#3}
20615   \cs_set:Npe \__peek_true_aux:w
20616   {
20617     \exp_not:N \group_align_safe_end:
20618     \exp_not:n {#4}
20619   }
20620   \cs_set_eq:NN \__peek_true:w #1
20621   \cs_set:Npe \__peek_false:w
20622   {
20623     \exp_not:N \group_align_safe_end:
20624     \exp_not:n {#5}
20625   }
20626   \peek_after:Nw #2
20627 }

```

(End of definition for `\__peek_token_generic_aux:NNNTF`.)

`\__peek_token_generic:NNTF`
`\__peek_token_remove_generic:NNTF`

For token removal there needs to be a call to the auxiliary function which does the work.

```

20628 \cs_new_protected:Npn \__peek_token_generic:NNTF
20629 { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
20630 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
20631 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
20632 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3
20633 { \__peek_token_generic:NNTF #1 #2 { } {#3} }
20634 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
20635 { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
20636 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
20637 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
20638 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3
20639 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End of definition for `\__peek_token_generic:NNTF` and `\__peek_token_remove_generic:NNTF`.)

`\__peek_execute_branches_meaning:`

The meaning test is straight forward.

```

20640 \cs_new:Npn \__peek_execute_branches_meaning:
20641 {
20642   \if_meaning:w \l__peek_token \l__peek_search_token
20643   \exp_after:wN \__peek_true:w
20644   \else:
20645     \exp_after:wN \__peek_false:w
20646   \fi:
20647 }

```

(End of definition for `\__peek_execute_branches_meaning:.`)

`\__peek_execute_branches_catcode:`
`\__peek_execute_branches_charcode:`
`\__peek_execute_branches_catcode_aux:`
`\__peek_execute_branches_catcode_auxii:N`
`\__peek_execute_branches_catcode_auxiii:`

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which are only given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using $\text{\TeX}$'s `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non- $\text{\LaTeX}$ 3 code) from blowing up. In the third case, `\l__peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

20648 \cs_new:Npn \__peek_execute_branches_catcode:
20649   { \if_catcode:w \__peek_execute_branches_catcode_aux: }
20650 \cs_new:Npn \__peek_execute_branches_charcode:
20651   { \if_charcode:w \__peek_execute_branches_catcode_aux: }
20652 \cs_new:Npn \__peek_execute_branches_catcode_aux:
20653   {
20654     \if_catcode:w \exp_not:N \l__peek_token \scan_stop:
20655       \exp_after:wN \exp_after:wN
20656       \exp_after:wN \__peek_execute_branches_catcode_auxiii:N
20657       \exp_after:wN \exp_not:N
20658     \else:
20659       \exp_after:wN \__peek_execute_branches_catcode_auxiii:
20660     \fi:
20661   }
20662 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
20663   {
20664     \exp_not:N #1
20665     \exp_after:wN \exp_not:N \l__peek_search_tl
20666     \exp_after:wN \__peek_true:w
20667   \else:
20668     \exp_after:wN \__peek_false:w
20669   \fi:
20670   #1
20671   }
20672 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
20673   {
20674     \exp_not:N \l__peek_token
20675     \exp_after:wN \exp_not:N \l__peek_search_tl
20676     \exp_after:wN \__peek_true:w
20677   \else:
20678     \exp_after:wN \__peek_false:w
20679   \fi:
20680   }

```

(End of definition for `\__peek_execute_branches_catcode:` and others.)

The public functions themselves cannot be defined using `\prg_new_protected_conditional:Npnn`. Instead, the TF, T, F variants are defined in terms of corresponding variants of

`\peek_catcode:N`
`\peek_catcode_remove:N`
`\peek_charcode:N`
`\peek_charcode_remove:N`
`\peek_meaning:N`
`\peek_meaning_remove:N`

`\__peek_token_generic:NNTF` or `\__peek_token_remove_generic:NNTF`, with first argument one of `\__peek_execute_branches_catcode:`, `\__peek_execute_branches_charcode:`, or `\__peek_execute_branches_meaning:`.

```

20681 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
20682 {
20683   \tl_map_inline:nn { { } { _remove } }
20684   {
20685     \tl_map_inline:nn { { TF } { T } { F } }
20686     {
20687       \cs_new_protected:cpe { peek_ #1 ##1 :N ####1 }
20688       {
20689         \exp_not:c { __peek_token ##1 _generic:NN ####1 }
20690         \exp_not:c { __peek_execute_branches_ #1 : }
20691       }
20692     }
20693   }
20694 }

```

(End of definition for `\peek_catcode:NNTF` and others. These functions are documented on page 212.)

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `\__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\__peek_use_none_delimit_by_s_stop:w` cleans up, and we call `\__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `\__peek_true:w` or `\__peek_false:w` as appropriate. Here, there is no `<search token>`, so we feed a dummy `\scan_stop:` to the `\__peek_token_generic:NNTF` function.

```

20695 \group_begin:
20696   \cs_set_protected:Npn \__peek_tmp:w #1 \s__peek_stop
20697   {
20698     \cs_new_protected:Npn \__peek_execute_branches_N_type:
20699     {
20700       \if_int_odd:w
20701         \if_catcode:w \exp_not:N \l_peek_token { \c_zero_int \fi:
20702         \if_catcode:w \exp_not:N \l_peek_token } \c_zero_int \fi:
20703         \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
20704         \c_one_int
20705       \exp_after:wN \__peek_N_type:w
20706       \token_to_meaning:N \l_peek_token
20707       \s__peek_mark \__peek_N_type_aux:nnw
20708       #1 \s__peek_mark \__peek_use_none_delimit_by_s_stop:w
20709       \s__peek_stop
20710       \exp_after:wN \__peek_true:w
20711     \else:
20712       \exp_after:wN \__peek_false:w

```

```

20713         \fi:
20714     }
20715     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \s__peek_mark ##3
20716     { ##3 {##1} {##2} }
20717 }
20718 \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \s__peek_stop
20719 \group_end:
20720 \cs_new_protected:Npn \__peek_N_type_aux:nw #1 #2 #3 \fi:
20721 {
20722     \fi:
20723     \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
20724     { \__peek_true:w }
20725     { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
20726 }
20727 \cs_new_protected:Npn \peek_N_type:TF
20728 {
20729     \__peek_token_generic:NNTF
20730     \__peek_execute_branches_N_type: \scan_stop:
20731 }
20732 \cs_new_protected:Npn \peek_N_type:T
20733 { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
20734 \cs_new_protected:Npn \peek_N_type:F
20735 { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }

```

(End of definition for \peek_N_type:TF and others. This function is documented on page 213.)

```

20736 </code>

```

Chapter 68

l3prop implementation

The following test files are used for this code: *m3prop001*, *m3prop002*, *m3prop003*, *m3prop004*, *m3show001*.

```
20737 <*code>
20738 <@@=prop>
```

With the (default) flat data storage, a property list is a macro whose top-level expansion is of the form

```
\s__prop \__prop_chk:w \__prop_pair:wn <key1> \s__prop {<value1>}
...
\__prop_pair:wn <keyn> \s__prop {<valuen>}
```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), `\__prop_chk:w` produces a suitable error if the property list is used directly in the input stream, and `\__prop_pair:wn` can be used to map through the property list.

With the linked data storage, each property list entry `<keyi>–<valuei>` is stored into a token list `\__prop <prefix> <keyi>`. The `<prefix>` is one or more characters (no spaces), constructed automatically only once, when the property list is initially declared. The control sequence name does not conform to standard naming for variables because (1) this is an internal control sequence, not really a `expl3` variable; (2) keeping track of the scope `l` or `g` throughout all functions would be a pretty big mess, especially if users accidentally mix local and global use (we would have to always check for such mistakes, rather than only checking when suitable debug options are set); (3) shorter control sequence names use less memory and are quicker in case of hash collisions, which may matter since we are using many control sequences.

We need to enable mapping through such a property list, but without storing a list of all entries anywhere: this is achieved by making each of these token lists also store a pointer to the next entry. To enable efficient deletion, the token lists also store a pointer to the previous entry. This means we have a doubly-linked list. To avoid having to special-case the two ends of the doubly-linked list when deleting entries, we include as a zeroth entry in the doubly-linked list the property list variable itself, and we include as an $(n + 1)$ -th entry in the doubly-linked list an end-pointer `\__prop <prefix>` (no trailing space, so it differs from an empty key). The space before `<prefix>` ensures there is no collision with other `l3prop` internal functions, even if we have very many linked property lists being defined.

The property list variable itself is a token list of the form

```
\__prop_flatten:w \__prop <prefix> \s__prop {\<prefix>} \__prop <prefix> <key1>
```

Here, `\__prop_flatten:w` serves as an efficiently recognized marker, and when `f`-expanded it is tasked with fully unpacking the property list into the same form as the default data storage so as to ease conversion. The `<prefix>` is used when looking up an entry. The token list `\__prop <prefix>` (see below) contains a pointer to the last key to help insert a new entry. The pointer to `<key1>` is needed to start a mapping. The token list labeled by `<keyi>` is of the form

```
\use_none:n \__prop <prefix> <keyi-1> \__prop_pair:wn <keyi> \s__-
prop {\<valuei>} \__prop <prefix> <keyi+1>
```

where the pointer to `<keyi-1>` is needed when deleting the `<keyi>`. Expanding this will run `\__prop_pair:wn` on the `<keyi>`–`<valuei>` pair (for speed, `<keyi>` is kept as explicit tokens rather than slowly extracting it from a control sequence name), then move on to the next key, thus mapping through the whole list. The mapping is ended upon expanding `\__prop <prefix>`, which is the token list

```
\use_none:n \__prop <prefix> <keyn>
```

Let us think about deleting the `<keyi>`. We need to update the `<keyi-1>` and `<keyi+1>` to point to each other instead of `<keyi>`. To edit the corresponding token lists, it is important that `\__prop <prefix> <keyi>` be at the “same place” in the token lists also in the boundary cases $i = 1$ or $i = n$, namely as the second token, or as the second argument after `\s__prop`.

68.1 Internal auxiliaries

`\__prop_tmp:w` Scratch macro, defined as needed, for instance to save `\__prop_pair:wn` when concatenating.

```
20739 \cs_new_eq:NN \__prop_tmp:w ?
```

(End of definition for `\__prop_tmp:w`.)

`\l__prop_tmp_tl` Token list used in various places: for the prefix; when converting from flat to linked props; and to store the new key–value pair inserted by `\prop_put:Nnn`.

```
20740 \tl_new:N \l__prop_tmp_tl
```

(End of definition for `\l__prop_tmp_tl`.)

`\s__prop_mark` Internal scan marks.

```
20741 \scan_new:N \s__prop_mark
```

```
20742 \scan_new:N \s__prop_stop
```

(End of definition for `\s__prop_mark` and `\s__prop_stop`.)

`\q__prop_recursion_tail` Internal recursion quarks.

```
20743 \quark_new:N \q__prop_recursion_tail
```

```
20744 \quark_new:N \q__prop_recursion_stop
```

(End of definition for `\q__prop_recursion_tail` and `\q__prop_recursion_stop`.)

```

\__prop_if_recursion_tail_stop:n Functions to query recursion quarks.
\__prop_if_recursion_tail_stop:o
20745 \__kernel_quark_new_test:N \__prop_if_recursion_tail_stop:n
20746 \cs_generate_variant:Nn \__prop_if_recursion_tail_stop:n { o }

(End of definition for \__prop_if_recursion_tail_stop:n and \__prop_if_recursion_tail_stop:o.)

```

68.2 Structure of a property list

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
20747 \scan_new:N \s__prop
```

(End of definition for `\s__prop`.)

`\__prop_chk:w` This removes the flat property list from the input stream and complains about a bad use of a property list. Since property lists do not have an end-marker, we slowly peek ahead in a loop. Speed does not matter since this is for an error situation. While `\__prop_pair:wn` does not keep a fixed definition, it always includes the internal `\s__prop` in its argument specification, so that there is no risk of accidentally picking up a public token instead of `\__prop_pair:wn` when doing a meaning test. We collect the keys and values to produce a more useful error message.

```

20748 \cs_new_protected:Npn \__prop_chk:w { \__prop_chk_loop:nw { } }
20749 \cs_new_protected:Npn \__prop_chk_loop:nw #1
20750 {
20751   \peek_meaning:NTF \__prop_pair:wn
20752   { \__prop_chk_get:nw {#1} }
20753   { \msg_error:nne { prop } { misused } {#1} }
20754 }
20755 \cs_new_protected:Npn \__prop_chk_get:nw #1 \__prop_pair:wn #2 \s__prop #3
20756 { \__prop_chk_loop:nw { #1 , ~ {#2} = { \tl_to_str:n {#3} } } }

```

(End of definition for `\__prop_chk:w`, `\__prop_chk_loop:nw`, and `\__prop_chk_get:nw`.)

`\__prop_pair:wn` Used as `\__prop_pair:wn <key> \s__prop {<item>}` for both storage types, this internal token starts each key–value pair in the property list. This default definition is changed globally by any mapping function, so there is not much point trying to make it an error. Instead, the error is produced by `\__prop_chk:w`.

```
20757 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2 { }
```

(End of definition for `\__prop_pair:wn`.)

`\__prop_flatten:w` We implement here the fact that f-expanding a linked property list gives a flat property list. Leaving a linked property list in the input stream will turn it into a flat property list so that the error implemented by `\__prop_chk:w` will correctly be triggered.

```

20758 \cs_new_protected:Npn \__prop_flatten:w #1 \s__prop #2#3
20759 { \use:e { \__prop_flatten_aux:N #3 } }

```

(End of definition for `\__prop_flatten:w`.)

`__prop_flatten:N` The main function `__prop_flatten:N` receives a linked property list and flattens it.
`__prop_flatten_aux:w` The auxiliary `__prop_flatten_aux:N` receives a pointer to the first key and flattens
`__prop_flatten_aux:N` the linked property list into a flat property list. This is only restricted-expandable
`__prop_flatten_loop:w` as it involves mapping through all of the property list's entries starting from $\langle key_1 \rangle$.
The looping function `__prop_flatten_loop:w` removes `use_none:n` and a backwards
pointer #2, leaves the key-value pair for `use:e` to receive, and calls itself again after
expanding the next key's token list. Its argument #3 is empty, except at the end where
it is the `use_none:nnnn` appearing in the definition of `__prop_flatten_aux:N`, which
ends the loop.

```

20760 \cs_new:Npn __prop_flatten:N #1
20761 { \exp_after:wN __prop_flatten_aux:w #1 }
20762 \cs_new:Npn __prop_flatten_aux:w #1 \s__prop #2 { __prop_flatten_aux:N }
20763 \cs_new:Npn __prop_flatten_aux:N #1
20764 {
20765   \s__prop __prop_chk:w
20766   \exp_after:wN __prop_flatten_loop:w #1
20767   \use_none:nnnn __prop_pair:wn \s__prop { }
20768 }
20769 \cs_new:Npn __prop_flatten_loop:w #1#2#3 __prop_pair:wn #4 \s__prop #5
20770 {
20771   #3
20772   \exp_not:n { __prop_pair:wn #4 \s__prop {#5} }
20773   \exp_after:wN __prop_flatten_loop:w
20774 }

```

(End of definition for `__prop_flatten:N` and others.)

`g__prop_prefix_int` Used to assign prefixes for each linked property list. It is converted to base `c__prop_`
`c__prop_basis_int` `basis_int`, then each digit is converted to a character, starting at ! (the character after
space).

```

20775 \int_new:N g__prop_prefix_int
20776 \int_const:Nn c__prop_basis_int { \c_max_char_int - '\! }

```

(End of definition for `g__prop_prefix_int` and `c__prop_basis_int`.)

`__prop_next_prefix:` Store in `l__prop_tmp_tl` the conversion of `g__prop_prefix_int` to characters, and
`__prop_to_prefix:n` increment this integer for use in the next linked property list. No need to optimize since
this is only used when declaring the property list the first time. The aim here is to make
this string as short as we can, given the range of distinct characters available. This speeds
up the work of `cs:w ... cs_end:` that looks up keys in the hash table.

```

20777 \cs_new_protected:Npn __prop_next_prefix:
20778 {
20779   \tl_set:Nc l__prop_tmp_tl
20780     { __prop_to_prefix:n { g__prop_prefix_int } }
20781   \int_gincr:N g__prop_prefix_int
20782 }
20783 \cs_new:Npn __prop_to_prefix:n #1
20784 {
20785   \int_compare:nNnTF {#1} > c__prop_basis_int
20786   {
20787     \exp_args:Nf __prop_to_prefix:n
20788       { \int_div_truncate:nn {#1} c__prop_basis_int }
20789     \exp_args:Nf __prop_to_prefix:n

```



```

20790         { \int_mod:nn {#1} \c__prop_basis_int }
20791     }
20792     { \char_generate:nn { '\! + #1 } { 12 } }
20793 }

```

(End of definition for `\__prop_next_prefix:` and `\__prop_to_prefix:n`.)

`\__prop_if_flat:N` We could either test for the presence of `\__prop_chk:w` (flat property list) or of `\__prop_flatten:w` (linked property list). We make the second choice; this way props that are accidentally `\relax` are treated as they were before. The auxiliary receives `\use_i:nn` or `\use_ii:nn` as #3. As a transitional fix we avoid erroring in case the prop is undefined (the `\exp_after:wN` is omitted in that case, taking the flat branch).

```

20794 \cs_new:Npn \__prop_if_flat:NTF #1
20795 {
20796     \prop_if_exist:NT #1
20797     \exp_after:wN \__prop_if_flat_aux:w #1
20798     \s__prop_mark \use_ii:nn
20799     \__prop_flatten:w \s__prop_mark \use_i:nn \s__prop_stop
20800 }
20801 \cs_new:Npn \__prop_if_flat_aux:w
20802     #1 \__prop_flatten:w #2 \s__prop_mark #3 #4 \s__prop_stop {#3}

```

(End of definition for `\__prop_if_flat:N` and `\__prop_if_flat_aux:w`.)

68.3 Allocation and initialization

`\c_empty_prop` An empty flat prop.

```

20803 \tl_const:Nn \c_empty_prop { \s__prop \__prop_chk:w }

```

(End of definition for `\c_empty_prop`. This variable is documented on page 228.)

`\prop_new:N` Flat property lists are initialized with the value `\c_empty_prop`.

```

\prop_new:c
20804 \cs_new_protected:Npn \prop_new:N #1
20805 {
20806     \__kernel_chk_if_free_cs:N #1
20807     \cs_gset_eq:NN #1 \c_empty_prop
20808 }
20809 \cs_generate_variant:Nn \prop_new:N { c }

```

(End of definition for `\prop_new:N`. This function is documented on page 220.)

`\prop_new_linked:N` The auxiliary is used in `\prop_make_linked:N`. For linked property lists, get a new prefix in `\l__prop_tmp_tl`, then use it to set up the internal structure: the last token in #1 is usually a pointer to the first key, which is here the end-pointer. That end-pointer has a pointer to the previous key (usually the last key), which is the variable #1 itself that begins the doubly-linked list.

```

\prop_new_linked:c
\__prop_new_linked:N
20810 \cs_new_protected:Npn \prop_new_linked:N #1
20811 {
20812     \__kernel_chk_if_free_cs:N #1
20813     \__prop_new_linked:N #1
20814 }
20815 \cs_new_protected:Npn \__prop_new_linked:N #1
20816 {

```

```

20817     \__prop_next_prefix:
20818     \cs_gset_nopar:Npe #1
20819     {
20820         \__prop_flatten:w
20821         \exp_not:c { \__prop ~ \l__prop_tmp_tl }
20822         \s__prop { \l__prop_tmp_tl }
20823         \exp_not:c { \__prop ~ \l__prop_tmp_tl }
20824     }
20825     \cs_gset_nopar:cpe { \__prop ~ \l__prop_tmp_tl }
20826     {
20827         \exp_not:N \use_none:n
20828         \exp_not:N #1
20829     }
20830 }
20831 \cs_generate_variant:Nn \prop_new_linked:N { c }

```

(End of definition for `\prop_new_linked:N` and `\__prop_new_linked:N`. This function is documented on page 220.)

\prop_clear:N Clearing a flat property list is like declaring it anew, simply setting it equal to `\c_empty_prop`. For linked property lists we must clear all of the variables storing individual keys, which requires a loop. At each step of the loop, `\__prop_clear_loop:Nw` receives `\cs_(g)set_eq:NN`, `\use_none:n`, the backwards pointer, an empty #4 (except at the end of the loop), and the key–value pair #5=#6 which we disregard. The looping auxiliary undefines the previous key’s token list (this includes the main token list, but that is fine because it is restored at the end) and calls itself after expanding the next key’s token list. The loop ends when #4 is `\use_none:nnnn`. After the loop, `\__prop_clear:wNNN` correctly sets up the main variable #6 and the end-pointer #1. Importantly, this is done using `\cs_(g)set_nopar:Npe` and `\exp_not:n` because the almost-equivalent `\tl_set:Nn` would complain in debug mode about the fact that the main variable is undefined at this stage. Importantly, `\__prop_clear_entries:NN` is used in the implementation of `\prop_set_eq:NN`.

```

20832 \cs_new_protected:Npn \prop_clear:N
20833 { \__prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe }
20834 \cs_generate_variant:Nn \prop_clear:N { c }
20835 \cs_new_protected:Npn \prop_gclear:N
20836 { \__prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
20837 \cs_generate_variant:Nn \prop_gclear:N { c }
20838 \cs_new_protected:Npn \__prop_clear:NNN #1#2#3
20839 {
20840     \__prop_if_flat:NTF #3
20841     { #1 #3 \c_empty_prop }
20842     { \exp_after:wN \__prop_clear:wNNN #3 #1 #2 #3 }
20843 }
20844 \cs_new_protected:Npn \__prop_clear:wNNN
20845     \__prop_flatten:w #1 \s__prop #2#3#4#5#6
20846 {
20847     \__prop_clear_entries:NN #4 #3
20848     #5 #6 { \exp_not:n { \__prop_flatten:w #1 \s__prop {#2} #1 } }
20849     #5 #1 { \exp_not:n { \use_none:n #6 } }
20850 }
20851 \cs_new_protected:Npn \__prop_clear_entries:NN #1#2
20852 {

```

```

20853 \exp_after:wN \_prop_clear_loop:Nw \exp_after:wN #1 #2
20854 \use_none:nnnn \_prop_pair:wn \s__prop { }
20855 }
20856 \cs_new_protected:Npn \_prop_clear_loop:Nw
20857 #1#2#3#4 \_prop_pair:wn #5 \s__prop #6
20858 {
20859 #1 #3 \tex_undefined:D
20860 #4
20861 \exp_after:wN \_prop_clear_loop:Nw
20862 \exp_after:wN #1
20863 }

```

(End of definition for `\prop_clear:N` and others. These functions are documented on page 220.)

```

\prop_clear_new:N A simple variation of the token list functions.
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c
\prop_clear_new_linked:N
\prop_clear_new_linked:c
\prop_gclear_new_linked:N
\prop_gclear_new_linked:c
20864 \cs_new_protected:Npn \prop_clear_new:N #1
20865 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
20866 \cs_generate_variant:Nn \prop_clear_new:N { c }
20867 \cs_new_protected:Npn \prop_gclear_new:N #1
20868 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
20869 \cs_generate_variant:Nn \prop_gclear_new:N { c }
20870 \cs_new_protected:Npn \prop_clear_new_linked:N #1
20871 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new_linked:N #1 } }
20872 \cs_generate_variant:Nn \prop_clear_new_linked:N { c }
20873 \cs_new_protected:Npn \prop_gclear_new_linked:N #1
20874 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new_linked:N #1 } }
20875 \cs_generate_variant:Nn \prop_gclear_new_linked:N { c }

```

(End of definition for `\prop_clear_new:N` and others. These functions are documented on page 220.)

```

\prop_set_eq:NN If both variables are accidentally the same variable (or equal flat property lists, as it
\prop_set_eq:cN turns out) we do nothing, otherwise the following code would lose all entries. If the
\prop_set_eq:Nc target variable #3 is a flat prop, either copy directly or flatten before copying. If it is a
\prop_set_eq:cc linked prop, we must clear it, then go through the entries in #4 to add them to #3.
\prop_gset_eq:NN
\prop_gset_eq:cN
\prop_gset_eq:Nc
\prop_gset_eq:cc
\_prop_set_eq:NNNN
\_prop_set_eq:wNNNN
\_prop_set_eq:nNnNN
\_prop_set_eq_loop:NNnw
\_prop_set_eq_end:w
20876 \cs_new_protected:Npn \prop_set_eq:NN
20877 { \_prop_set_eq:NNNN \cs_set_eq:NN \cs_set_nopar:Npe }
20878 \cs_generate_variant:Nn \prop_set_eq:NN { Nc , cN , cc }
20879 \cs_new_protected:Npn \prop_gset_eq:NN
20880 { \_prop_set_eq:NNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
20881 \cs_generate_variant:Nn \prop_gset_eq:NN { Nc , cN , cc }
20882 \cs_new_protected:Npn \_prop_set_eq:NNNN #1#2#3#4
20883 {
20884 \cs_if_eq:NNF #3#4
20885 {
20886 \_prop_if_flat:NTF #3
20887 {
20888 \_prop_if_flat:NTF #4
20889 { #1 #3 #4 }
20890 { #2 #3 { \_prop_flatten:N #4 } }
20891 }
20892 { \exp_after:wN \_prop_set_eq:wNNNN #3 #1#2#3#4 }
20893 }
20894 }
20895 \cs_new_protected:Npn \_prop_set_eq:wNNNN

```

```

20896   \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
20897   {
20898     \__prop_clear_entries:NN #4 #3
20899     \exp_args:Nf \__prop_set_eq:nNnNN {#7} #1 {#2} #5 #6
20900   }

```

We have used that `f`-expanding either type of prop gives a flat prop. At this stage `\__prop_set_eq:nNnNN` receives the second variable as a flat prop, the end-pointer, the prefix, the suitable `\cs_(g)set_nopar:Npe` assignment, and the first variable itself. Remove the leading `\s__prop` and `\__prop_chk:w` with `\use_i:nnn`, then start the loop.

```

20901 \cs_new_protected:Npn \__prop_set_eq:nNnNN #1#2#3#4#5
20902 {
20903   \use_i:nnn
20904   {
20905     \__prop_set_eq_loop:NNnw #5 #4 {#3}
20906     \__prop_flatten:w #2 \s__prop {#3}
20907   }
20908   #1
20909   \use_none:n \__prop_pair:wn ? \s__prop
20910 }

```

The looping function receives the current pointer `#1` (initially the variable itself), the defining function `#2` and the prefix `#3`, then a partial definition `#4` (which in later stages includes the backwards pointer), followed by the current value as `\s__prop {#5}`. It seeks the next key `#7` to construct in `\l__prop_tmp_tl` the next pointer `\__prop <prefix> <next key>` (the argument `#6` is empty, except at the end of the loop, where it is `\use_none:n` in such a way as to delete the `<space>` and `<next key>`). Then the token list (current pointer) `#1` is set-up to contain the partial definition and current value, as well as the newly constructed next pointer. After a line responsible for correctly ending the loop with `\__prop_set_eq_end:w`, we loop, setting up the next definition, which starts with `\use_none:n` and a backwards pointer to `#1` followed by the `<next key>` `#7` and so on.

```

20911 \cs_new_protected:Npn \__prop_set_eq_loop:NNnw
20912   #1#2#3#4 \s__prop #5#6 \__prop_pair:wn #7 \s__prop
20913 {
20914   \tl_set:Nx \l__prop_tmp_tl { \exp_not:c { __prop ~ #3 #6 ~ #7 } }
20915   #2 #1 { \exp_not:n { #4 \s__prop {#5} } \exp_not:o \l__prop_tmp_tl }
20916   \use_none:n #6 \__prop_set_eq_end:w
20917   \exp_after:wN \__prop_set_eq_loop:NNnw \l__prop_tmp_tl #2 {#3}
20918   \use_none:n #1 \__prop_pair:wn #7 \s__prop
20919 }

```

The end-code picks up what is needed to correctly assign the last token list (the end pointer), which is simply `\use_none:n \__prop_<prefix><space><keyn>`.

```

20920 \cs_new_protected:Npn \__prop_set_eq_end:w
20921   \exp_after:wN \__prop_set_eq_loop:NNnw #1#2#3
20922   \use_none:n #4#5 \s__prop
20923 {
20924   \exp_after:wN #2 \l__prop_tmp_tl { \exp_not:n { \use_none:n #4 } }
20925 }

```

(End of definition for `\prop_set_eq:NN` and others. These functions are documented on page 220.)

`\prop_make_flat:N` The only interesting case is when given a linked prop. Clear the linked property list using `\__prop_clear:wNNN` with local assignments (it does not matter since we are at

the outermost group level, and `\cs_set_eq:NN` is very slightly faster than its global version. Then store the contents (expanded preventively by `\exp_args:NNf`) with an assignment `\cs_set_nopar:Npe` that does not perform `l3debug` checks.

```

20926 \cs_new_protected:Npn \prop_make_flat:N #1
20927 {
20928   \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
20929   {
20930     \__prop_if_flat:NTF #1 { }
20931     { \exp_args:NNf \__prop_make_flat:Nn #1 {#1} }
20932   }
20933   {
20934     \msg_error:nnee { prop } { inner-make }
20935     { \token_to_str:N \prop_make_flat:N } { \token_to_str:N #1 }
20936   }
20937 }
20938 \cs_generate_variant:Nn \prop_make_flat:N { c }
20939 \cs_new_protected:Npn \__prop_make_flat:Nn #1#2
20940 {
20941   \exp_after:wN \__prop_clear:wNNN #1 \cs_set_eq:NN \cs_set_nopar:Npe #1
20942   \cs_set_nopar:Npe #1 { \exp_not:n {#2} }
20943 }

```

(End of definition for `\prop_make_flat:N` and `\prop_make_flat:c \__prop_make_flat:Nn`. This function is documented on page 221.)

`\prop_make_linked:N`
`\prop_make_linked:c`
`\__prop_make_linked:Nn`

The only interesting case is when given a flat prop. We expand the contents for later use. Then `\__prop_new_linked:N` disregards that previous value of `#1` and initializes the linked prop. We can then use an auxiliary `\__prop_set_eq:wNNNN` underlying `\prop_set_eq:NN`, with the prop contents saved as `\l__prop_tmp_tl`. That step is a bit unsafe, as `\l__prop_tmp_tl` (really, a flat prop here) is used within `\__prop_set_eq:wNNNN` itself, but it is in fact expanded early enough to be ok.

```

20944 \cs_new_protected:Npn \prop_make_linked:N #1
20945 {
20946   \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
20947   {
20948     \__prop_if_flat:NTF #1
20949     { \exp_args:NNo \__prop_make_linked:Nn #1 {#1} } { }
20950   }
20951   {
20952     \msg_error:nnee { prop } { inner-make }
20953     { \token_to_str:N \prop_make_linked:N } { \token_to_str:N #1 }
20954   }
20955 }
20956 \cs_generate_variant:Nn \prop_make_linked:N { c }
20957 \cs_new_protected:Npn \__prop_make_linked:Nn #1#2
20958 {
20959   \__prop_new_linked:N #1
20960   \tl_set:Nn \l__prop_tmp_tl {#2}
20961   \exp_after:wN \__prop_set_eq:wNNNN #1
20962   \cs_set_eq:NN \cs_set_nopar:Npe #1 \l__prop_tmp_tl
20963 }

```

(End of definition for `\prop_make_linked:N` and `\__prop_make_linked:Nn`. This function is documented on page 221.)

`\l_tmpa_prop` We can now initialize the scratch variables.

```

20964 \prop_new:N \l_tmpa_prop
20965 \prop_new:N \l_tmpb_prop
20966 \prop_new:N \g_tmpa_prop
20967 \prop_new:N \g_tmpb_prop

```

(End of definition for `\l_tmpa_prop` and others. These variables are documented on page 228.)

`\prop_concat:NNN` The basic strategy is to copy the first variable into the target, then loop through the
`\prop_concat:ccc` second variable, calling `\prop_(g)put:Nnn` on each item. To avoid running the `l3debug`
`\prop_gconcat:NNN` scope checks on each of these steps, we use the auxiliaries that underly `\prop_set_eq:NN`
`\prop_gconcat:ccc` and `\prop_put:Nnn`, whose syntax is a bit unwieldy. We work directly with the target
`\__prop_concat:NNNNN` `prop #3` as a scratch space, because copying over from a temporary variable to `#3` would
`\__prop_concat:nNNN` be slow in the linked case. If `#5` is `#3` itself we have to be careful not to lose the data, and
we even take the opportunity to skip the copying step completely. To keep the correct
version of the duplicate keys we use the code underlying `\prop_put_if_not_in:Nnn`,
which involves passing `\use_none:nnn` to the auxiliary instead of nothing. There is no
need to check for the case where `#3` is equal to `#4` because in that case `\prop_(g)set_`
`eq:NN #3 #4` (or rather the underlying auxiliary) is correctly set up to do no needless
work.

```

20968 \cs_new_protected:Npn \prop_concat:NNN
20969 { \__prop_concat:NNNNN \cs_set_eq:NN \cs_set_nopar:Npe }
20970 \cs_generate_variant:Nn \prop_concat:NNN { ccc }
20971 \cs_new_protected:Npn \prop_gconcat:NNN
20972 { \__prop_concat:NNNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
20973 \cs_generate_variant:Nn \prop_gconcat:NNN { ccc }
20974 \cs_new_protected:Npn \__prop_concat:NNNNN #1#2#3#4#5
20975 {
20976   \cs_if_eq:NNTF #3 #5
20977   { \__prop_concat:nNNN \use_none:nnn #2 #3 #4 }
20978   {
20979     \__prop_set_eq:NNNN #1 #2 #3 #4
20980     \__prop_concat:nNNN { } #2 #3 #5
20981   }
20982 }
20983 \cs_new_protected:Npn \__prop_concat:nNNN #1#2#3#4
20984 {
20985   \cs_gset_eq:NN \__prop_tmp:w \__prop_pair:wn
20986   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop
20987   { \__prop_put:nNNnn {#1} #2 #3 {##1} }
20988   \exp_last_unbraced:Nf \use_none:nn #4
20989   \cs_gset_eq:NN \__prop_pair:wn \__prop_tmp:w
20990 }

```

(End of definition for `\prop_concat:NNN` and others. These functions are documented on page 222.)

`\prop_put_from_keyval:Nn` The core is a call to `\keyval_parse:nnn`, with an error message `\__prop_missing_eq:n`
`\prop_put_from_keyval:cn` for entries without `=`, and a call to (essentially) `\prop_(g)put:Nnn` for valid key–value
`\prop_gput_from_keyval:Nn` pairs. To avoid repeated scope checks (and errors) when `l3debug` is active, we instead use
`\prop_gput_from_keyval:cn` the auxiliary underlying `\prop_put:Nnn`. Because blank keys are valid here, in contrast
`\__prop_from_keyval:nn` to `l3keys`, we set and restore `\l__kernel_keyval_allow_blank_keys_bool`. The key–
`\__prop_from_keyval:Nnn` value argument may be quite large so we avoid reading it until it is really necessary.
`\__prop_missing_eq:n`

```

20991 \cs_new_protected:Npn \prop_put_from_keyval:Nn #1

```

```

20992 { \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_set_nopar:Npe #1 } }
20993 \cs_generate_variant:Nn \prop_put_from_keyval:Nn { c }
20994 \cs_new_protected:Npn \prop_gput_from_keyval:Nn #1
20995 { \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_gset_nopar:Npe #1 } }
20996 \cs_generate_variant:Nn \prop_gput_from_keyval:Nn { c }
20997 \cs_new_protected:Npn \_prop_from_keyval:nn
20998 {
20999   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
21000   { \_prop_from_keyval:Nnn \c_true_bool }
21001   { \_prop_from_keyval:Nnn \c_false_bool }
21002 }
21003 \cs_new_protected:Npn \_prop_from_keyval:Nnn #1#2#3
21004 {
21005   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool \c_true_bool
21006   \keyval_parse:nnn \_prop_missing_eq:n {#2} {#3}
21007   \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool #1
21008 }
21009 \cs_new_protected:Npn \_prop_missing_eq:n
21010 { \msg_error:nnn { prop } { prop-keyval } }

```

(End of definition for `\prop_put_from_keyval:Nn` and others. These functions are documented on page 223.)

`\prop_set_from_keyval:Nn`
`\prop_set_from_keyval:cn`
`\prop_gset_from_keyval:Nn`
`\prop_gset_from_keyval:cn`

Just empty the prop (with the auxiliary underlying `\prop_clear:N` to avoid l3debug problems) and push key–value entries using `\prop_(g)put_from_keyval:Nn`.

```

21011 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1
21012 {
21013   \_prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe #1
21014   \prop_put_from_keyval:Nn #1
21015 }
21016 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
21017 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1
21018 {
21019   \_prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe #1
21020   \prop_gput_from_keyval:Nn #1
21021 }
21022 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }

```

(End of definition for `\prop_set_from_keyval:Nn` and `\prop_gset_from_keyval:Nn`. These functions are documented on page 221.)

`\prop_const_from_keyval:Nn`
`\prop_const_from_keyval:cn`
`\prop_const_linked_from_keyval:Nn`
`\prop_const_linked_from_keyval:cn`

For both flat and linked constant props, we create #1 then use the same auxiliary as for `\prop_gput_from_keyval:Nn`. It is most natural to use the already packaged `\prop_gput:Nnn`, but that would mean doing an assignment on a supposedly constant property list. To avoid errors when l3debug is activated, we use the auxiliary underlying `\prop_gput:Nnn`.

```

21023 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1
21024 {
21025   \prop_new:N #1
21026   \_prop_from_keyval:nn { \_prop_put:nNNnn { } \cs_gset_nopar:Npe #1 }
21027 }
21028 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
21029 \cs_new_protected:Npn \prop_const_linked_from_keyval:Nn #1
21030 {

```

```

21031 \prop_new_linked:N #1
21032 \__prop_from_keyval:nn { \__prop_put:nNnn { } \cs_gset_nopar:Npe #1 }
21033 }
21034 \cs_generate_variant:Nn \prop_const_linked_from_keyval:Nn { c }

```

(End of definition for `\prop_const_from_keyval:Nn` and `\prop_const_linked_from_keyval:Nn`. These functions are documented on page 221.)

68.4 Accessing data in property lists

Accessing/deleting/adding entries is mostly done by `\__prop_split:NnTFn`, which must be fast because it is used in many `l3prop` functions. Its syntax is as follows.

```

\__prop_split:NnTFn <property list> {<key>}
{<true code>} {<false code>} {<link code>}

```

If the `<property list>` uses the linked data storage, then it runs the `<link code>`, otherwise it does as follows.

It splits the `<property list>` at the `<key>`, giving three token lists: the `<entries before>` the `<key>`, the `<value>` associated with the `<key>` and the `<entries after>` the `<key>`. Both the `<entries before>` and the `<entries after>` can be empty or consist of some number of consecutive entries `\__prop_pair:wn <keyi> \s__prop {<valuei>}`. If the `<key>` is present in the `<property list>` then the `<true code>` is left in the input stream, with #1, #2, and #3 replaced by the `<entries before>`, `<value>`, and `<entries after>`. If the `<key>` is not present in the `<property list>` then the `<false code>` is left in the input stream. Only the `<true code>` is used in the replacement text of a macro defined internally, which requires ## doubling.

```

\__prop_split:NnTFn
\__prop_split_aux:nNnTFn
\__prop_split_test:wn
\__prop_split_flat:w
\__prop_split_linked:w
\__prop_split_wrong:Nw

```

The aim is to distinguish four cases: a flat prop that contains the given `<key>`, a flat prop that does not contain it, a linked prop, and an invalid prop. The last case includes those that are set to `\relax` by c-expansion, as well as unrelated token list variables since these unfortunately used to “work” in earlier implementations. In the first three cases we run the T, F, and n arguments, and in the last case we raise an error, set the variable to a known state (empty prop), and run the F code (some conditionals such as `\prop_pop:NnNnTF` otherwise blow up pretty badly).

The first distinction between these cases is done by `\__prop_split_test:wn`, which looks for the argument after `\s__prop`. For a flat prop it will be `\__prop_chk:w`, which leads to running `\__prop_split_flat:w`, explained below. For a linked prop it is the prefix, consisting of characters, so we end up running `\__prop_split_linked:w`, which cleans up and selects the aforementioned n argument. For invalid props, or rather, variables that do not contain `\s__prop`, the argument includes `\fi:`, and we end up calling `\__prop_split_wrong:Nw`, which calls `\prop_show:N` to raise a detailed error stating how the variable is wrong.

Let us return to `\__prop_split_flat:w`. This function is defined dynamically as

```

\cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \s__prop_mark #4 #5 \s__prop_stop
{ #4 {<true code>} }

```


Its job is to seek the $\langle \text{key} \rangle$ in the property list (known to be flat at this stage) by using an argument #1 delimited essentially by that key. If indeed the variable contained the $\langle \text{key} \rangle$, then #1 is the $\langle \text{extract}_1 \rangle$ before the key–value pair, #2 is the $\langle \text{value} \rangle$ associated with the $\langle \text{key} \rangle$, #3 is the $\langle \text{extract}_2 \rangle$ after the key–value pair, #4 is $\backslash \text{use\_i:nnn}$, and we run $\backslash \text{use\_i:nnn} \{ \langle \text{true code} \rangle \} \{ \langle \text{false code} \rangle \} \{ \langle \text{link code} \rangle \}$, selecting the $\langle \text{true code} \rangle$. Otherwise, the whole property list together with $\backslash \text{s\_prop\_mark} \backslash \text{use\_i:nnn}$ is taken in as #1, then #2 is some tokens $? \backslash \text{fi:} \backslash \text{__prop\_split\_wrong:Nw} \langle \text{variable} \rangle$ that were only useful in the case of invalid props, #3 is empty, and most importantly #4 is $\backslash \text{use\_ii:nnn}$. This command selects the $\langle \text{false code} \rangle$.

Note that we define $\backslash \text{__prop\_split\_flat:w}$ in all cases even though it is only used in the flat case. Indeed, to avoid taking in the whole property list (which may be large) as an argument more than strictly necessary, we would have to keep the $\langle \text{true code} \rangle$ positioned before the expansion of the prop variable in order to use it in the definition. The only way to do that is to store it using an assignment so we might as well just perform the assignment that we can actually use in the flat case.

```

21035 \cs_new_protected:Npn \__prop_split:NnTFn #1#2
21036 {
21037   \exp_after:wN \__prop_split_aux:nNTFn
21038   \exp_after:wN { \tl_to_str:n {#2} } #1
21039 }
21040 \cs_new_protected:Npn \__prop_split_aux:nNTFn #1#2#3
21041 {
21042   \cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w ##1
21043   \__prop_pair:wn #1 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
21044   { ##4 {#3} }
21045   \exp_after:wN \__prop_split_test:wn #2 \s__prop_mark \use_i:nnn
21046   \__prop_pair:wn #1 \s__prop { ? \fi: \__prop_split_wrong:Nw #2 }
21047   \s__prop_mark \use_ii:nnn
21048   \s__prop_stop
21049 }
21050 \cs_new:Npn \__prop_split_flat:w { }
21051 \cs_new_protected:Npn \__prop_split_test:wn #1 \s__prop #2
21052 {
21053   \if_meaning:w \__prop_chk:w #2 \exp_after:wN \__prop_split_flat:w \fi:
21054   \__prop_split_linked:w
21055 }
21056 \cs_new_protected:Npn \__prop_split_linked:w #1 \s__prop_stop #2#3 {#3}
21057 \cs_new_protected:Npn \__prop_split_wrong:Nw #1#2 \s__prop_stop #3#4
21058 {
21059   \prop_show:N #1
21060   \cs_gset_eq:NN #1 \c_empty_prop
21061   #3
21062 }

```

(End of definition for $\backslash \text{__prop\_split:NnTFn}$ and others.)

$\backslash \text{prop\_get:NnN}$ Here we implement both $\backslash \text{prop\_get:NnN}$ and its branching version through $\backslash \text{__prop\_get:NnnTF}$. It receives the prop and key, followed by an assignment used when the value is found, $\langle \text{true code} \rangle$ to run after the assignment, and some fall-back $\langle \text{false code} \rangle$ for absent values. It relies on $\backslash \text{__prop\_split:NnTFn}$. For a flat prop, the first four arguments of $\backslash \text{__prop\_split:NnTFn}$ are used, and run either the assignment #3{##3} and $\langle \text{true code} \rangle$ #4, or the $\langle \text{false code} \rangle$ #5.

$\backslash \text{prop\_get:NvN}$

$\backslash \text{prop\_get:NeN}$

$\backslash \text{prop\_get:NoN}$

$\backslash \text{prop\_get:NxN}$

$\backslash \text{prop\_get:cnN}$

$\backslash \text{prop\_get:cVN}$

$\backslash \text{prop\_get:cvN}$

$\backslash \text{prop\_get:ceN}$

$\backslash \text{prop\_get:coN}$

$\backslash \text{prop\_get:cxN}$

$\backslash \text{prop\_get:cnc}$

$\backslash \text{prop\_get:NnNTF}$

$\backslash \text{prop\_get:NvNTF}$

$\backslash \text{prop\_get:NvNTF}$

$\backslash \text{prop\_get:NeNTF}$

```

21063 \cs_new_protected:Npn \prop_get:NnN #1#2#3
21064 {
21065   \__prop_get:NnnTF #1 {#2}
21066   { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
21067 }
21068 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , Ne , c , cV , cv , ce }
21069 \cs_generate_variant:Nn \prop_get:NnN { No , Nx , co , cx }
21070 \cs_generate_variant:Nn \prop_get:NnN { cnc }
21071 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
21072 {
21073   \__prop_get:NnnTF #1 {#2}
21074   { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
21075 }
21076 \prg_generate_conditional_variant:Nnn \prop_get:NnN
21077 { NV , Nv , Ne , c , cV , cv , ce } { T , F , TF }
21078 \prg_generate_conditional_variant:Nnn \prop_get:NnN
21079 { No , Nx , co , cx } { T , F , TF }
21080 \prg_generate_conditional_variant:Nnn \prop_get:NnN
21081 { cnc } { T , F , TF }
21082 \cs_new_protected:Npn \__prop_get:NnnTF #1#2#3#4#5
21083 {
21084   \__prop_split:NnTFn #1 {#2}
21085   { #3 {##2} #4 }
21086   {#5}
21087   { \exp_after:wN \__prop_get_linked:w #1 {#2} {#3} {#4} {#5} }
21088 }

```

For a linked prop we must work a bit: `\__prop_get_linked:w` is followed by the expansion of the prop, then by four brace groups: the key #4, the assignment code #5, `<true code>` #6, and `<false code>` #7. If the key is present, its value is stored in the token list `\__prop_#2~#4`. If that token list exists, `\__prop_get_linked_aux:w` gets called followed by the expansion of that token list and we grab as #2 the value associated to that key, which we feed to the assignment code and follow-up code. If the key is absent the token list can be `\undefined` or `\relax`. In both cases `\__prop_get_linked_aux:w` finds an empty brace group as #2, `\use_none:n` as #4 and the `<false code>` as #5. Note that we made `\__prop_get_linked:w` and subsequent auxiliaries expandable, because they are also used in `\prop_item:Nn`.

```

21089 \cs_new:Npn \__prop_get_linked:w
21090   \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
21091 {
21092   \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
21093   \exp_after:wN \exp_after:wN \exp_after:wN \__prop_get_linked_aux:w
21094   \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \exp_after:wN \cs_end:
21095   \else:
21096     \exp_after:wN \__prop_get_linked_aux:w
21097   \fi:
21098   \s__prop_mark {#5} {#6}
21099   \s__prop { } \s__prop_mark \use_none:n {#7}
21100   \s__prop_stop
21101 }
21102 \cs_new:Npn \__prop_get_linked_aux:w
21103   #1 \s__prop #2 #3 \s__prop_mark #4 #5 #6 \s__prop_stop { #4 {#2} #5 }

```

(End of definition for `\prop_get:NnN` and others. These functions are documented on page 223.)

\prop_item:Nn Getting the value corresponding to a key in a flat property list in an expandable fashion simply uses **\prop_map_tokens:Nn** to go through the property list. The auxiliary **\prop_item:Nv** **__prop_item:nnn** receives the search string #1, the key #2 and the value #3 and returns as appropriate.

\prop_item:cn 21104 **\cs_new:Npn \prop_item:Nn #1#2**
\prop_item:cV 21105 {
\prop_item:co 21106 **__prop_if_flat:Ntf #1**
\prop_item:ce 21107 {
__prop_item:nnn 21108 **\exp_args:NNo \prop_map_tokens:Nn #1**
21109 {
21110 **\exp_after:wN __prop_item:nnn**
21111 **\exp_after:wN { \tl_to_str:n {#2} }**
21112 }
21113 }
21114 { **\exp_after:wN __prop_get_linked:w #1 {#2} \exp_not:n { } { } }**
21115 }
21116 **\cs_new:Npn __prop_item:nnn #1#2#3**
21117 {
21118 **\str_if_eq:eeT {#1} {#2}**
21119 { **\prop_map_break:n { \exp_not:n {#3} }** }
21120 }
21121 **\cs_generate_variant:Nn \prop_item:Nn { NV , No , Ne , c , cV , co , ce }**

(End of definition for **\prop_item:Nn** and **__prop_item:nnn**. This function is documented on page 224.)

68.5 Removing data from property lists

__prop_pop:NnNnTF This auxiliary is used by both the **\prop_pop** family and the **\prop_remove** family of functions. It receives a $\langle prop \rangle$ and a $\{ \langle key \rangle \}$, three assignment functions (**\tl_set:Nn** **\cs_set_eq:NN** **\cs_set_nopar:Npe** or their global versions), then $\{ \langle code \rangle \}$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$.

For a flat prop, split it. If the $\langle key \rangle$ is there, reconstruct the rest of the prop from the two extracts **##2** **##4** and assign using **\tl_(g)set:Nn**, then run $\langle code \rangle$ $\{ \langle value \rangle \}$ with the $\langle value \rangle$ found, and run the $\langle true code \rangle$. If the $\langle key \rangle$ is absent, run the $\langle false code \rangle$.

For a linked prop, the removal is done by **__prop_pop_linked:wnNnTF**, which removes the key–value pair from the doubly-linked list and runs its last three arguments $\{ \langle code \rangle \}$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$ depending on whether the key–value is found, in the same way as for flat props.

21122 **\cs_new_protected:Npn __prop_pop:NnNnTF #1#2#3#4#5#6#7**
21123 {
21124 **__prop_split:NnTFn #1 {#2}**
21125 {
21126 **#4 #1 { \exp_not:n { \s__prop __prop_chk:w ##1 ##3 } }**
21127 **#5 {##2}**
21128 **#6**
21129 }
21130 **{#7}**
21131 {
21132 **\exp_after:wN __prop_pop_linked:wnNnTF #1 {#2}**
21133 **#3 #4 {#5} {#6} {#7}**
21134 }

21135 }

The next auxiliary `\__prop_pop_linked:wnNNnTF`, together with the `NNNn` auxiliary, checks if the key is present in the $\langle linked\ prop \rangle$, then the corresponding value (if present) is passed as a braced argument to the $\langle code \rangle$ and the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ is run as appropriate. Before that, there are also three assignments: the token lists for the previous key and next key are made to point to each other, cf. `\__prop_pop_linked:w`, and the token list for the given key is made undefined.

```

21136 \cs_new_protected:Npn \__prop_pop_linked:wnNNnTF
21137   \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
21138   {
21139     \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
21140     \exp_after:wN \__prop_pop_linked:NNNn
21141     \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
21142     #5 #6 {#7}
21143   \else:
21144     \exp_after:wN \use_iii:nnn
21145   \fi:
21146   \use_i:nn
21147 }
21148 \cs_new_protected:Npn \__prop_pop_linked:NNNn #1#2#3#4
21149 {
21150   \if_meaning:w \scan_stop: #1
21151     \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
21152   \else:
21153     \exp_after:wN \__prop_pop_linked:w #1 #1 #2 #3 {#4}
21154   \fi:
21155 }
21156 \cs_new_protected:Npn \__prop_pop_linked:w
21157   \use_none:n #1#2 \s__prop #3#4#5#6#7#8
21158 {
21159   #6 #5 \tex_undefined:D
21160   #7 #1
21161   {
21162     \exp_after:wN \__prop_pop_linked_prev:w #1
21163     \exp_not:N #4
21164   }
21165   #7 #4
21166   {
21167     \exp_not:n { \use_none:n #1 }
21168     \exp_not:f { \exp_after:wN \__prop_pop_linked_next:w #4 }
21169   }
21170   #8 {#3}
21171 }
21172 \cs_new:Npn \__prop_pop_linked_prev:w #1 \s__prop #2#3
21173   { \exp_not:n { #1 \s__prop {#2} } }
21174 \cs_new:Npn \__prop_pop_linked_next:w \use_none:n #1 { \exp_stop_f: }

```

(End of definition for `\__prop_pop:NnNNnTF` and others.)

`\prop_remove:Nn` Deleting from a property relies on `\__prop_pop:NnNNnTF`. The three assignment functions are suitably local or global. The last three arguments are `\use_none:n` and two empty brace groups: if the key is found we get `\use_none:n { $\langle key \rangle$ }  $\langle empty \rangle$` , which ex-

`\prop_remove:NV`

`\prop_remove:Ne`

`\prop_remove:cn`

`\prop_remove:cV`

`\prop_remove:ce`

`\prop_gremove:Nn`

`\prop_gremove:NV`

`\prop_gremove:Ne`

`\prop_gremove:cn`

`\prop_gremove:cV`

`\prop_gremove:ce`

pands to nothing, and otherwise we just get *<empty>*. The auxiliary takes care of actually removing the entry from the prop.

```

21175 \cs_new_protected:Npn \prop_remove:Nn #1#2
21176 {
21177   \__prop_pop:NnNnTF #1 {#2}
21178   \cs_set_eq:NN \cs_set_nopar:Npe
21179   \use_none:n { } { }
21180 }
21181 \cs_new_protected:Npn \prop_gremove:Nn #1#2
21182 {
21183   \__prop_pop:NnNnTF #1 {#2}
21184   \cs_gset_eq:NN \cs_gset_nopar:Npe
21185   \use_none:n { } { }
21186 }
21187 \cs_generate_variant:Nn \prop_remove:Nn { NV , Ne , c , cV , ce }
21188 \cs_generate_variant:Nn \prop_gremove:Nn { NV , Ne , c , cV , ce }

```

(End of definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 224.)

<code>\prop_pop:NnN</code> <code>\prop_pop:NVN</code> <code>\prop_pop:NoN</code> <code>\prop_pop:cnN</code> <code>\prop_pop:cVN</code> <code>\prop_pop:coN</code> <code>\prop_gpop:NnN</code> <code>\prop_gpop:NVN</code> <code>\prop_gpop:NoN</code> <code>\prop_gpop:cnN</code> <code>\prop_gpop:cVN</code> <code>\prop_gpop:coN</code> <code>\prop_pop:NnNTF</code> <code>\prop_pop:NVNTF</code> <code>\prop_pop:NoNTF</code> <code>\prop_pop:cnNTF</code> <code>\prop_pop:cVNTF</code> <code>\prop_pop:coNTF</code> <code>\prop_gpop:NnNTF</code> <code>\prop_gpop:NVNTF</code> <code>\prop_gpop:NoNTF</code> <code>\prop_gpop:cnNTF</code> <code>\prop_gpop:cVNTF</code> <code>\prop_gpop:coNTF</code>	Popping a value is almost the same, but the value found is kept. For the non-branching version, we additionally set the target token list to <code>\q_no_value</code> , while for the branching version we must produce <code>\prg_return_true:</code> or <code>\prg_return_false:</code> . <pre> 21189 \cs_new_protected:Npn \prop_pop:NnN #1#2#3 21190 { 21191 __prop_pop:NnNnTF #1 {#2} 21192 \cs_set_eq:NN \cs_set_nopar:Npe 21193 { \tl_set:Nn #3 { } } { \tl_set:Nn #3 { \q_no_value } } 21194 } 21195 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3 21196 { 21197 __prop_pop:NnNnTF #1 {#2} 21198 \cs_gset_eq:NN \cs_gset_nopar:Npe 21199 { \tl_set:Nn #3 { } } { \tl_set:Nn #3 { \q_no_value } } 21200 } 21201 \cs_generate_variant:Nn \prop_pop:NnN { NV , No } 21202 \cs_generate_variant:Nn \prop_pop:NnN { c , cV , co } 21203 \cs_generate_variant:Nn \prop_gpop:NnN { NV , No } 21204 \cs_generate_variant:Nn \prop_gpop:NnN { c , cV , co } 21205 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF } 21206 { 21207 __prop_pop:NnNnTF #1 {#2} 21208 \cs_set_eq:NN \cs_set_nopar:Npe 21209 { \tl_set:Nn #3 { } } \prg_return_true: \prg_return_false: 21210 } 21211 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF } 21212 { 21213 __prop_pop:NnNnTF #1 {#2} 21214 \cs_gset_eq:NN \cs_gset_nopar:Npe 21215 { \tl_set:Nn #3 { } } \prg_return_true: \prg_return_false: 21216 } 21217 \prg_generate_conditional_variant:Nnn \prop_pop:NnN 21218 { NV , No , c , cV , co } { T , F , TF } 21219 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN </pre>

```
21220 { NV , No , c , cV , co } { T , F , TF }
```

(End of definition for `\prop_pop:NnN` and others. These functions are documented on page 223.)

68.6 Adding data to property lists

`\prop_put:Nnn` All of the `\prop_(g)put(_if_new):Nnn` functions are based on the same auxiliary, which receives `<code>` and an “assignment”, followed by `<prop> {<key>} {<new value>}`. The assignment `\cs_(g)set_nopar:Npe` is the primitive assignment without any checking: in the case of linked props it is applied to individual pieces of the linked prop, which are typically not yet defined. Debugging the scope of the variable is done at a higher level by letting `l3debug` change `\prop_put:Nnn` and friends. This allows other `l3prop` commands to directly call the underlying auxiliary to skip this checking step and avoid getting multiple error messages for the same error. The `<code>` (empty for `put` and `\use_none:nnn` for `put_if_not_in`) is placed before the assignment in cases where the key is already present, in order to suppress the assignment in the `put_if_not_in` case.

```
21221 \cs_new_protected:Npn \prop_put:Nnn
21222 { \__prop_put:nNnn { } \cs_set_nopar:Npe }
21223 \cs_new_protected:Npn \prop_gput:Nnn
21224 { \__prop_put:nNnn { } \cs_gset_nopar:Npe }
21225 \cs_new_protected:Npn \prop_put_if_not_in:Nnn
21226 { \__prop_put:nNnn \use_none:nnn \cs_set_nopar:Npe }
21227 \cs_new_protected:Npn \prop_gput_if_not_in:Nnn
21228 { \__prop_put:nNnn \use_none:nnn \cs_gset_nopar:Npe }
21229 \cs_generate_variant:Nn \prop_put:Nnn
21230 {
21231     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
21232     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
21233 }
21234 \cs_generate_variant:Nn \prop_put:Nnn
21235 { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
21236 \cs_generate_variant:Nn \prop_put:Nnn
21237 {
21238     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
21239     cv , cvV , cvv , cve , ce , ceV , cev , cee
21240 }
21241 \cs_generate_variant:Nn \prop_put:Nnn
21242 { cno , co , coo , cnx , cVx , cxV , cxx }
21243 \cs_generate_variant:Nn \prop_gput:Nnn
21244 {
21245     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
21246     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
21247 }
21248 \cs_generate_variant:Nn \prop_gput:Nnn
21249 { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
21250 \cs_generate_variant:Nn \prop_gput:Nnn
21251 {
21252     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
21253     cv , cvV , cvv , cve , ce , ceV , cev , cee
21254 }
21255 \cs_generate_variant:Nn \prop_gput:Nnn
21256 { cno , co , coo , cnx , cVx , cxV , cxx }
```

`\prop_gput:Nnn`

`\prop_gput:NnV`

`\prop_gput:Nnv`

`\prop_gput:Nne`

`\prop_gput:NVn`

`\prop_gput:NVV`

`\prop_gput:NVv`

`\prop_gput:NVe`

```

21257 \cs_generate_variant:Nn \prop_put_if_not_in:Nnn
21258 {
21259     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
21260     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
21261     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
21262     cv , cvV , cvv , cve , ce , ceV , cev , cee
21263 }
21264 \cs_generate_variant:Nn \prop_gput_if_not_in:Nnn
21265 {
21266     NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
21267     Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
21268     c , cnV , cnv , cne , cV , cVV , cVv , cVe ,
21269     cv , cvV , cvv , cve , ce , ceV , cev , cee
21270 }

```

Since the true branch of `\__prop_split:NnTFn` is used as the replacement text of an internal macro, and since the `<key>` and new `<value>` may contain arbitrary tokens, it is not safe to include them in the argument of `\__prop_split:NnTFn`. We thus start by storing in `\l__prop_tmp_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `\__prop_split:NnTFn`. For a flat prop, if the `<key>` was absent, append the new key–value to the list; otherwise concatenate the extracts `##2` and `##4` with the new key–value pair `\l__prop_tmp_tl`. The updated entry is placed at the same spot as the original `<key>` in the property list, preserving the order of entries. For a linked prop, call `\__prop_put_linked:wnN`, which constructs the control sequence in which we will place the new value. If it matches `\scan_stop:` then the key was not yet there and we add it using `\__prop_put_linked_new:w`, otherwise it was already there and we use `\__prop_put_linked_old:w`.

```

21271 \cs_new_protected:Npn \__prop_put:nNnn #1#2#3#4#5
21272 {
21273     \tl_set:Nn \l__prop_tmp_tl
21274     {
21275         \exp_not:N \__prop_pair:wn \tl_to_str:n {#4}
21276         \s__prop { \exp_not:n {#5} }
21277     }
21278     \__prop_split:NnTFn #3 {#4}
21279     {
21280         #1 #2 #3
21281         {
21282             \s__prop \__prop_chk:w \exp_not:n {##1}
21283             \l__prop_tmp_tl \exp_not:n {##3}
21284         }
21285     }
21286     { #2 #3 { \exp_not:o {#3} \l__prop_tmp_tl } }
21287     { \exp_after:wN \__prop_put_linked:wnnN #3 {#4} {#1} #2 }
21288 }
21289 \cs_new_protected:Npn \__prop_put_linked:wnnN
21290 \__prop_flatten:w #1 \s__prop #2#3#4
21291 {
21292     \exp_after:wN \__prop_put_linked:NNnN
21293     \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
21294     #1
21295 }
21296 \cs_new_protected:Npn \__prop_put_linked:NNnN #1#2#3#4
21297 {

```

```

21298 \if_meaning:w \scan_stop: #1
21299 \exp_after:wN \__prop_put_linked_new:w #2 #1 #2 #4
21300 \else:
21301 \exp_after:wN \__prop_put_linked_old:w #1 { #3 #4 #1 }
21302 \fi:
21303 }

```

To add a new entry, `\__prop_put_linked_new:w` receives the expansion of the end-pointer, namely `\use_none:n` (*last key pointer*), followed by the new key pointer #2, the end pointer #3, and an assignment function #4. Set up the doubly-linked list in the order #1, #2, #3, placing the key-value pair `\l__prop_tmp_tl` in #2. To replace an old entry, `\__prop_put_linked_old:w` receives the expansion of that entry, and it reassigns it (#5) using the assignment #6, by simply replacing the payload #2 `\s__prop` #3 by `\l__prop_tmp_tl`.

```

21304 \cs_new_protected:Npn \__prop_put_linked_new:w
21305 \use_none:n #1#2#3#4
21306 {
21307   #4 #1
21308   {
21309     \exp_after:wN \__prop_pop_linked_prev:w #1
21310     \exp_not:N #2
21311   }
21312   #4 #2
21313   {
21314     \exp_not:n { \use_none:n #1 }
21315     \l__prop_tmp_tl
21316     \exp_not:N #3
21317   }
21318   #4 #3 { \exp_not:n { \use_none:n #2 } }
21319 }
21320 \cs_new_protected:Npn \__prop_put_linked_old:w
21321 \use_none:n #1#2 \s__prop #3#4#5
21322 {
21323   #5
21324   {
21325     \exp_not:n { \use_none:n #1 }
21326     \l__prop_tmp_tl
21327     \exp_not:N #4
21328   }
21329 }

```

(End of definition for `\prop_put:Nnn` and others. These functions are documented on page 222.)

68.7 Property list conditionals

```

\prop_if_exist_p:N Copies of the cs functions defined in l3basics.
\prop_if_exist_p:c 21330 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
\prop_if_exist:NTF 21331 { TF , T , F , p }
\prop_if_exist:cTF 21332 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
21333 { TF , T , F , p }

```

(End of definition for `\prop_if_exist:NTF`. This function is documented on page 224.)

`\prop_if_empty_p:N` A flat property list is empty if it matches `\c_empty_prop`. A linked property list is empty if its second token (the end pointer) and last token (the first key pointer) are equal. There cannot be false positives because the end pointer takes the form `\use_none:n <pointer>` while the other pointers have more elaborate structure. The subtle code branch here is when a non-empty flat property list is given: then `\__prop_if_empty:w` reads the whole property list as #1 and #2, #3, #4 are 2, 3, 4, respectively.

```

21334 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
21335 {
21336   \if_meaning:w #1 \c_empty_prop
21337   \prg_return_true:
21338   \else:
21339     \exp_after:wN \__prop_if_empty_return:w #1
21340     \__prop_flatten:w 2 \s__prop 34 \s__prop_stop
21341   \fi:
21342 }
21343 \cs_new:Npn \__prop_if_empty_return:w
21344 #1 \__prop_flatten:w #2 \s__prop #3#4#5 \s__prop_stop
21345 {
21346   \if_meaning:w #2 #4
21347   \prg_return_true:
21348   \else:
21349     \prg_return_false:
21350   \fi:
21351 }
21352 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
21353 { c } { p , T , F , TF }

```

(End of definition for `\prop_if_empty:N` and `\__prop_if_empty_return:w`. This function is documented on page 225.)

`\prop_if_in_p:Nn` For a linked prop, use `\__prop_get_linked:w` to look up whether the control sequence constructed from the prefix and the sought-after key exists; this auxiliary calls `\use_none:n {<value>}` `\prg_return_true:` if the key is found, and otherwise `\prg_return_false:`. For a flat prop, testing expandably if a key is there requires to go through the key-value pairs one by one. This is rather slow, and a faster test would be

`\prop_if_in_p:Nv`
`\prop_if_in_p:Ne`
`\prop_if_in_p:No`
`\prop_if_in_p:cn`
`\prop_if_in_p:cV`
`\prop_if_in_p:ce`
`\prop_if_in_p:co`
`\prop_if_in:NnTF`
`\prop_if_in:NvTF`
`\prop_if_in:NeTF`
`\prop_if_in:NoTF`
`\prop_if_in:cnTF`
`\prop_if_in:cVTF`
`\prop_if_in:ceTF`
`\prop_if_in:coTF`
`\__prop_if_in_flat:nnn`

```

\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
{
  \@@_split:NnTFn #1 {#2}
  { \prg_return_true: }
  { \prg_return_false: }
  { ... }
}

```

but `\__prop_split:NnTFn` is non-expandable. Instead, we use `\prop_map_tokens:Nn` to compare the search key to each key in turn using `\str_if_eq:ee`, which is expandable.

```

21354 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
21355 {
21356   \__prop_if_flat:NTF #1
21357   {
21358     \exp_after:wN \prop_map_tokens:Nn \exp_after:wN #1
21359     {
21360       \exp_after:wN \__prop_if_in_flat:nnn

```

```

21361         \exp_after:wN { \tl_to_str:n {#2} }
21362     }
21363     \prg_return_false:
21364 }
21365 {
21366     \exp_after:wN \__prop_get_linked:w #1 {#2}
21367     \use_none:n \prg_return_true: \prg_return_false:
21368 }
21369 }
21370 \cs_new:Npn \__prop_if_in_flat:nnn #1#2#3
21371 {
21372     \str_if_eq:eeT {#1} {#2}
21373     { \prop_map_break:n { \use_i:nn \prg_return_true: } }
21374 }
21375 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
21376 { NV , Ne , No , c , cV , ce , co } { p , T , F , TF }

```

(End of definition for `\prop_if_in:NnTF` and `\__prop_if_in_flat:nnn`. This function is documented on page 225.)

68.8 Mapping over property lists

`\prop_map_function:NN` We first f-expand to flatten #1 in case it was a linked list. The `\use_i:nnn` removes the leading `\s__prop \__prop_chk:w` of the flattened prop. The even-numbered arguments of `\__prop_map_function:Nw` are keys, hence have string catcodes, except at the end where they are `\fi: \prop_map_break:.` The `\fi:` ends the `\if_false: #⟨even⟩ \fi:` construction and we jump out of the loop. No need for any quark test.

```

21377 \cs_new:Npn \prop_map_function:NN #1#2
21378 {
21379     \exp_last_unbraced:Nnf
21380     \use_i:nnn { \__prop_map_function:Nw #2 } #1
21381     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21382     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21383     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21384     \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21385     \prg_break_point:Nn \prop_map_break: { }
21386 }
21387 \cs_new:Npn \__prop_map_function:Nw #1
21388     \__prop_pair:wn #2 \s__prop #3
21389     \__prop_pair:wn #4 \s__prop #5
21390     \__prop_pair:wn #6 \s__prop #7
21391     \__prop_pair:wn #8 \s__prop #9
21392 {
21393     \if_false: #2 \fi: #1 {#2} {#3}
21394     \if_false: #4 \fi: #1 {#4} {#5}
21395     \if_false: #6 \fi: #1 {#6} {#7}
21396     \if_false: #8 \fi: #1 {#8} {#9}
21397     \__prop_map_function:Nw #1
21398 }
21399 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End of definition for `\prop_map_function:NN` and `\__prop_map_function:Nw`. This function is documented on page 226.)

\prop_map_inline:Nn Mapping in line requires a nesting level counter. Store the current definition of `\__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `\__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

```

21400 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
21401 {
21402   \cs_gset_eq:cN
21403   { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
21404   \int_gincr:N \g__kernel_prg_map_int
21405   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
21406   \exp_last_unbraced:Nf \use_none:nn #1
21407   \prg_break_point:Nn \prop_map_break:
21408   {
21409     \int_gdecr:N \g__kernel_prg_map_int
21410     \cs_gset_eq:Nc \__prop_pair:wn
21411     { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn }
21412   }
21413 }
21414 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End of definition for `\prop_map_inline:Nn`. This function is documented on page 226.)

\prop_map_tokens:Nn The mapping is very similar to `\prop_map_function:NN`. The odd construction `\use:n {#1}` allows #1 to contain any token without interfering with `\prop_map_break:.`
\prop_map_tokens:cn `\use:n {#1}` allows #1 to contain any token without interfering with `\prop_map_break:.`
__prop_map_tokens:nw The loop stops when the `<key>` between `\__prop_pair:wn` and `\s__prop` is `\fi: \prop_map_break:` instead of being a string.

```

21415 \cs_new:Npn \prop_map_tokens:Nn #1#2
21416 {
21417   \exp_last_unbraced:Nnf
21418   \use_i:nnn { \__prop_map_tokens:nw {#2} } #1
21419   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21420   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21421   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21422   \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
21423   \prg_break_point:Nn \prop_map_break: { }
21424 }
21425 \cs_new:Npn \__prop_map_tokens:nw #1
21426   \__prop_pair:wn #2 \s__prop #3
21427   \__prop_pair:wn #4 \s__prop #5
21428   \__prop_pair:wn #6 \s__prop #7
21429   \__prop_pair:wn #8 \s__prop #9
21430 {
21431   \if_false: #2 \fi: \use:n {#1} {#2} {#3}
21432   \if_false: #4 \fi: \use:n {#1} {#4} {#5}
21433   \if_false: #6 \fi: \use:n {#1} {#6} {#7}
21434   \if_false: #8 \fi: \use:n {#1} {#8} {#9}
21435   \__prop_map_tokens:nw {#1}
21436 }
21437 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End of definition for `\prop_map_tokens:Nn` and `\__prop_map_tokens:nw`. This function is documented on page 226.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.
`\prop_map_break:n`

```
21438 \cs_new:Npn \prop_map_break:
21439 { \prg_map_break:Nn \prop_map_break: { } }
21440 \cs_new:Npn \prop_map_break:n
21441 { \prg_map_break:Nn \prop_map_break: }
```

(End of definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 227.)

68.9 Uses of mapping over property lists

`\prop_count:N` Counting the key–value pairs in a property list is done using the same approach as for
`\prop_count:c` other count functions: turn each entry into a +1 then use integer evaluation to actually
`\__prop_count:nn` do the mathematics.

```
21442 \cs_new:Npn \prop_count:N #1
21443 {
21444   \int_eval:n
21445   {
21446     0
21447     \prop_map_function:NN #1 \__prop_count:nn
21448   }
21449 }
21450 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
21451 \cs_generate_variant:Nn \prop_count:N { c }
```

(End of definition for `\prop_count:N` and `\__prop_count:nn`. This function is documented on page 224.)

`\prop_to_keyval:N` Each property name and value pair will be returned in the form $\sqcup\{\langle name \rangle\}=\sqcup\{\langle value \rangle\}$.
`\__prop_to_keyval_exp_after:wN` As one of the main use cases for this macro is to pass the `\property list` on to a
`\__prop_to_keyval:nn` key–value parser, we have to make sure that the behavior is as good as possible. Using a
`\__prop_to_keyval:nnw` space before the opening brace we get the correct brace stripping behavior for most of the
key–value parsers available in L^AT_EX. Iterate over the `\property list` and remove the
leading comma afterwards. Only the value has to be protected in `\__kernel_exp_not:w`
as the property name is always a string. After the loop the leading comma is removed by
`\use_none:n` and afterwards `\__kernel_exp_not:w` eventually finds the opening brace
of its argument.

```
21452 \cs_new:Npn \prop_to_keyval:N #1
21453 {
21454   \__kernel_exp_not:w
21455   \prop_if_empty:NTF #1
21456   { {} }
21457   {
21458     \exp_after:wN \exp_after:wN \exp_after:wN
21459     {
21460       \tex_expanded:D
21461       {
21462         \exp_not:N \use_none:n
21463         \prop_map_function:NN #1 \__prop_to_keyval:nn
21464       }
21465     }
21466   }
21467 }
```

```

21468 \cs_new:Npn \__prop_to_keyval:nn #1#2
21469 { , ~ {#1} =~ { \__kernel_exp_not:w {#2} } }

```

(End of definition for `\prop_to_keyval:N` and others. This function is documented on page [224](#).)

68.10 Viewing property lists

```

\prop_show:N
\prop_show:c
\prop_log:N
\prop_log:c

```

Experience shows one source of problems is very hard to debug: when a data structure such as a `seq` or `prop` gets corrupted. In the past, `\prop_show:N` would in some cases happily show items of such a `prop`, even though other more demanding `!3prop` functions would choke. It is thus best to make `\prop_show:N` check very thoroughly the structure and flag issues, even though that is very painful for linked props. Throughout the code below, we strive to remain as safe as possible, but in the explanations we only state what the arguments are when the prop is correctly formed, rather than saying at every step that various arguments can be arbitrary junk, made safe by using `\tl_to_str:n` generously.

The general `\__kernel_chk_tl_type:NnnT` checks that its first argument is a token list, and if it is, then it `e`-expands its second argument and compares with the contents of its first argument. Thus, within this `e`-expansion it is safe to use `\__prop_if_flat:NTF` to check if the prop is flat or linked. In the flat case we simply reconstruct the expected structure using `\__prop_show_flat:w`, which loops through the prop and correctly turns all keys to strings for instance. In the linked case, we use `\__prop_show_linked:w`, which ensures the form `\__prop_flatten:w \__prop {<prefix> \s__prop {<prefix>} <rest>}`, where `<prefix>` is made into a string and `<rest>` cannot be a brace group or multiple tokens since `\__prop_show_linked:w` would in such cases give a different result from the original token list.

```

21470 \cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nneeee }
21471 \cs_generate_variant:Nn \prop_show:N { c }
21472 \cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nneeee }
21473 \cs_generate_variant:Nn \prop_log:N { c }
21474 \cs_new_protected:Npn \__prop_show:NN #1#2
21475 {
21476   \__kernel_chk_tl_type:NnnT #2 { prop }
21477   {
21478     \__prop_if_flat:NTF #2
21479     {
21480       \s__prop \__prop_chk:w
21481       \exp_after:wN \__prop_show_flat:w #2
21482       \s__prop { }
21483       \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
21484       \q__prop_recursion_stop
21485     }
21486     { \exp_after:wN \__prop_show_linked:w #2 \s__prop ! ? \s__prop_stop }
21487   }
21488   {
21489     \__prop_if_flat:NTF #2
21490     { \__prop_show_finally:NNn #1 #2 { flat } }
21491     {
21492       \tl_set:Nn \l__prop_tmp_tl { #1 #2 }
21493       \exp_after:wN \__prop_show_prepare:w #2 #2
21494     }
21495   }

```

```

21496 }
21497 \cs_new:Npn \__prop_show_flat:w #1 \__prop_pair:wn #2 \s__prop #3
21498 {
21499   \__prop_if_recursion_tail_stop:n {#2}
21500   \exp_not:N \__prop_pair:wn \tl_to_str:n {#2} \s__prop \exp_not:n { {#3} }
21501   \__prop_show_flat:w
21502 }
21503 \cs_new:Npn \__prop_show_linked:w #1 \s__prop #2#3#4 \s__prop_stop
21504 {
21505   \exp_not:N \__prop_flatten:w
21506   \exp_not:c { __prop ~ \tl_to_str:n {#2} }
21507   \s__prop { \tl_to_str:n {#2} }
21508   \exp_not:n {#3}
21509 }

```

For flat props we are done by using `\msg_show:nneeee` or `\msg_log:nneeee`. The auxiliary `\__prop_show_finally:NNn` is eventually also used in the linked case after some more tests. To avoid having to bring along the message function and the property list, we store them into `\l__prop_tmp_tl`.

```

21510 \cs_new_protected:Npn \__prop_show_finally:NNn #1#2#3
21511 {
21512   #1 { prop } { show }
21513   { \token_to_str:N #2 }
21514   { \prop_map_function:NN #2 \msg_show_item:nn }
21515   {#3} { }
21516 }

```

For linked props, we now know they have a reasonable form so that we are calling `\__prop_show_prepare:w \__prop_flatten:w \__prop <prefix> \s__prop {<prefix>} <token> <property list>`, and the task is to loop through the linked list and check integrity. We first set things up: the auxiliary `\__prop_tmp:w` will be in charge of checking that various tokens start with `\__prop <prefix>` (in the sense of string representations), and calling one of `\__prop_show_loop_key:wNNN`, `\__prop_show_end:NNN`, `\__prop_show_bad_name:NNN`.

```

21517 \cs_new_protected:Npn \__prop_show_prepare:w
21518   \__prop_flatten:w #1 \s__prop #2#3#4
21519 {
21520   \use:e
21521   {
21522     \cs_set_nopar:Npn \exp_not:N \__prop_tmp:w
21523       ##1 \token_to_str:N #1 ##2 \s__prop_mark ##3 \s__prop_stop
21524     {
21525       \exp_not:N \tl_if_empty:nTF {##1}
21526       {
21527         \exp_not:N \tl_if_head_is_space:nTF {##2}
21528         { \exp_not:N \exp_args:Nf \__prop_show_loop_key:wNNN }
21529         { \exp_not:N \tl_if_empty:nTF }
21530         {##2}
21531       }
21532       { \exp_not:N \use_ii:nn }
21533       \__prop_show_end:NNN
21534       \__prop_show_bad_name:NNN
21535     }
21536   }

```

```

21537 \exp_last_unbraced:NNNo \__prop_show_loop:NNw #1 #4 #4
21538 }

```

The loop will consist of calls to `\__prop_show_loop:NNw \__prop <prefix> <token> <expansion>`, where `<token>` is one of the items in the list, specifically the key container for `<keyi-1>` (starting at $i = 1$ with the property list variable itself), and `<expansion>` stands for the expansion of that token, which has already been checked, and takes the form `<junk> \s__prop {<value>} \__prop <prefix> <keyi>`. Thus, the loop auxiliary receives the prefix command as #1, and the $(i - 1)$ -th and i -th key containers as #2 and #5. Then `\__prop_tmp:w` checks that the name of the i -th key container is valid.

```

21539 \cs_new_protected:Npn \__prop_show_loop:NNw #1#2 #3 \s__prop #4#5
21540 {
21541   \exp_last_two_unbraced:Noo \__prop_tmp:w
21542   { \token_to_str:N #5 \s__prop_mark }
21543   { \token_to_str:N #1 \s__prop_mark \s__prop_stop }
21544   #1 #2 #5
21545 }

```

If the i -th key container has the wrong name we get `\__prop_show_bad_name:NNN \__prop <prefix> <previous container> <current container with bad name>`.

```

21546 \cs_new_protected:Npn \__prop_show_bad_name:NNN #1#2#3
21547 {
21548   \msg_error:nneeee { prop } { bad-link }
21549   { \tl_tail:N \l__prop_tmp_tl }
21550   { \token_to_str:N #2 }
21551   { \token_to_str:N #3 }
21552   { \token_to_str:N #1 }
21553 }

```

If the i -th key container has the name `\__prop <prefix>` (without space), it is the trailing one. We check that it is the right kind of macro to be a token list, and that it has the right contents `\use_none:n <previous container>`. If so, we are done checking everything, and we display the property list using the message function and property list name stored in `\l__prop_tmp_tl`. Note that we also use this `\l__prop_tmp_tl` in the type argument of `\__kernel_chk_tl_type:NnnT`, to build up the name “`<property list> prop entry`” used in error messages.

```

21554 \cs_new_protected:Npn \__prop_show_end:NNN #1#2#3
21555 {
21556   \__kernel_chk_tl_type:NnnT #3
21557   { \tl_tail:N \l__prop_tmp_tl prop~entry }
21558   { \exp_not:n { \use_none:n #2 } }
21559   {
21560     \exp_after:wN \__prop_show_finally:NNn
21561     \l__prop_tmp_tl { linked }
21562   }
21563 }

```

If the i -th container has a name `\__prop <prefix> <key>` (with a space before the key), then we have a call to `__prop_show_loop_key:wNNN {<key>} <junk12. (with an f-expansion to eliminate the space). The first argument is the <key> without a leading space, thanks to a judicious f-expansion earlier on. We check that the <current container> is a token list with the expected structure \use_none:n <previous container> __prop_pair:wn <string> \s__prop {<anything>} <single token>. The auxiliary __prop_show_flat:w`

is reused to produce the `\__prop_pair:wn` part, and the last token is produced by `\tl_item:Nn` (we don't waste a specialized auxiliary to speed that up). If the check succeed, move on to the next item.

```

21564 \cs_new_protected:Npn \__prop_show_loop_key:wNNN #1#2#3#4#5#6
21565 {
21566   \__kernel_chk_tl_type:NnnT #6
21567   { \tl_tail:N \l__prop_tmp_tl prop~entry }
21568   {
21569     \exp_not:n { \use_none:n #5 }
21570     \exp_after:wN \__prop_show_flat:w #6 \s__prop { }
21571     \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
21572     \q__prop_recursion_stop
21573     \tl_item:Nn #6 { -1 }
21574   }
21575   { \exp_last_unbraced:NNNo \__prop_show_loop:NNw #4 #6 #6 }
21576 }

```

(End of definition for `\prop_show:N` and others. These functions are documented on page [227](#).)

```

21577 </code>

```


Chapter 69

l3skip implementation

```
21578 <*code>
21579 <@@=dim>
```

69.1 Length primitives renamed

```
\if_dim:w Primitives renamed.
  \__dim_eval:w 21580 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
\__dim_eval_end: 21581 \cs_new_eq:NN \__dim_eval:w \tex_dimexpr:D
                21582 \cs_new_eq:NN \__dim_eval_end: \tex_relax:D

(End of definition for \if_dim:w, \__dim_eval:w, and \__dim_eval_end:.. This function is documented
on page 244.)
```

69.2 Internal auxiliaries

```
\s__dim_mark Internal scan marks.
\s__dim_stop 21583 \scan_new:N \s__dim_mark
              21584 \scan_new:N \s__dim_stop

(End of definition for \s__dim_mark and \s__dim_stop.)

\_dim_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
21585 \cs_new:Npn \_dim_use_none_delimit_by_s_stop:w #1 \s__dim_stop { }

(End of definition for \_dim_use_none_delimit_by_s_stop:w.)
```

69.3 Creating and initializing dim variables

```
\dim_new:N Allocating <dim> registers ...
\dim_new:c 21586 \cs_new_protected:Npn \dim_new:N #1
           21587 {
           21588   \__kernel_chk_if_free_cs:N #1
           21589   \cs:w newdimen \cs_end: #1
           21590 }
           21591 \cs_generate_variant:Nn \dim_new:N { c }
```

(End of definition for `\dim_new:N`. This function is documented on page 229.)

`\dim_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
21592 \cs_new_protected:Npn \dim_const:Nn #1#2
21593 {
21594   \dim_new:N #1
21595   \tex_global:D #1 = \dim_eval:n {#2} \scan_stop:
21596 }
21597 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End of definition for `\dim_const:Nn`. This function is documented on page 229.)

`\dim_zero:N` Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a $\text{\LaTeX} 2_{\epsilon}$ length). Besides, these functions are then simply copied for `\skip_zero:N` and related functions.

```
\dim_zero:c
\dim_gzero:N
\dim_gzero:c
21598 \cs_new_protected:Npn \dim_zero:N #1 { #1 = \c_zero_skip }
21599 \cs_new_protected:Npn \dim_gzero:N #1
21600 { \tex_global:D #1 = \c_zero_skip }
21601 \cs_generate_variant:Nn \dim_zero:N { c }
21602 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End of definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 229.)

`\dim_zero_new:N` Create a register if needed, otherwise clear it.

```
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c
21603 \cs_new_protected:Npn \dim_zero_new:N #1
21604 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
21605 \cs_new_protected:Npn \dim_gzero_new:N #1
21606 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
21607 \cs_generate_variant:Nn \dim_zero_new:N { c }
21608 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(End of definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 229.)

`\dim_if_exist_p:N` Copies of the cs functions defined in `l3basics`.

```
\dim_if_exist_p:c
\dim_if_exist:NTF
\dim_if_exist:cTF
21609 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
21610 { TF , T , F , p }
21611 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
21612 { TF , T , F , p }
```

(End of definition for `\dim_if_exist:NTF`. This function is documented on page 230.)

69.4 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a $\text{\LaTeX} 2_{\epsilon}$ length).

```
\dim_gset:Nn
\dim_gset:cn
\dim_gset:NV
\dim_gset:cV
21613 \cs_new_protected:Npn \dim_set:Nn #1#2
21614 { #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
```

```

21615 \cs_new_protected:Npn \dim_gset:Nn #1#2
21616 { \tex_global:D #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
21617 \cs_generate_variant:Nn \dim_set:Nn { NV , c , cV }
21618 \cs_generate_variant:Nn \dim_gset:Nn { NV , c , cV }

```

(End of definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 230.)

```

\dim_set_eq:NN All straightforward, with a \scan_stop: to deal with the case where #1 is (incorrectly)
\dim_set_eq:cN a skip.
\dim_set_eq:Nc
\dim_set_eq:cc
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc

```

```

21619 \cs_new_protected:Npn \dim_set_eq:NN #1#2
21620 { #1 = #2 \scan_stop: }
21621 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
21622 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
21623 { \tex_global:D #1 = #2 \scan_stop: }
21624 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }

```

(End of definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 230.)

```

\dim_add:Nn Using by here would slow things down just to detect nonsensical cases such as passing
\dim_add:cN \dimen 123 as the first argument. Using \scan_stop: deals with skip variables. Since
\dim_gadd:Nn debugging checks that the variable is correctly local/global, the global versions cannot
\dim_gadd:cN be defined as \tex_global:D followed by the local versions.

```

```

\dim_sub:Nn
\dim_sub:cN
\dim_gsub:Nn
\dim_gsub:cN

```

```

21625 \cs_new_protected:Npn \dim_add:Nn #1#2
21626 { \tex_advance:D #1 \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
21627 \cs_new_protected:Npn \dim_gadd:Nn #1#2
21628 {
21629   \tex_global:D \tex_advance:D #1
21630   \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
21631 }
21632 \cs_generate_variant:Nn \dim_add:Nn { c }
21633 \cs_generate_variant:Nn \dim_gadd:Nn { c }
21634 \cs_new_protected:Npn \dim_sub:Nn #1#2
21635 { \tex_advance:D #1 - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
21636 \cs_new_protected:Npn \dim_gsub:Nn #1#2
21637 {
21638   \tex_global:D \tex_advance:D #1
21639   -\__dim_eval:w #2 \__dim_eval_end: \scan_stop:
21640 }
21641 \cs_generate_variant:Nn \dim_sub:Nn { c }
21642 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End of definition for `\dim_add:Nn` and others. These functions are documented on page 230.)

69.5 Utilities for dimension calculations

`\__dim_sep:`

```

21643 \cs_new_eq:NN \__dim_sep: \__kernel_int_sep:

```

(End of definition for `\__dim_sep:.`)

\dim_abs:n Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading - if present.

__dim_abs:N

\dim_max:nn

\dim_min:nn

__dim_maxmin:wwN

```

21644 \cs_new:Npn \dim_abs:n #1
21645 {
21646   \exp_after:wN \__dim_abs:N
21647   \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
21648 }
21649 \cs_new:Npn \__dim_abs:N #1
21650 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
21651 \cs_new:Npn \dim_max:nn #1#2
21652 {
21653   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
21654   \dim_use:N \__dim_eval:w #1 \exp_after:wN \__dim_sep:
21655   \dim_use:N \__dim_eval:w #2 \__dim_sep:
21656   >
21657   \__dim_eval_end:
21658 }
21659 \cs_new:Npn \dim_min:nn #1#2
21660 {
21661   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
21662   \dim_use:N \__dim_eval:w #1 \exp_after:wN \__dim_sep:
21663   \dim_use:N \__dim_eval:w #2 \__dim_sep:
21664   <
21665   \__dim_eval_end:
21666 }
21667 \cs_new:Npn \__dim_maxmin:wwN #1 \__dim_sep: #2 \__dim_sep: #3
21668 {
21669   \if_dim:w #1 #3 #2 ~
21670   #1
21671   \else:
21672   #2
21673   \fi:
21674 }

```

(End of definition for \dim_abs:n and others. These functions are documented on page 230.)

\dim_ratio:nn With dimension expressions, something like 10 pt * (5 pt / 10 pt) does not work. Instead, the ratio part needs to be converted to an integer expression. Using \int_value:w forces everything into sp, avoiding any decimal parts.

__dim_ratio:n

```

21675 \cs_new:Npn \dim_ratio:nn #1#2
21676 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
21677 \cs_new:Npn \__dim_ratio:n #1
21678 { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End of definition for \dim_ratio:nn and __dim_ratio:n. This function is documented on page 231.)

69.6 Dimension expression conditionals

\dim_compare_p:nNn Simple comparison.

\dim_compare_p:vNn

\dim_compare_p:nNv

\dim_compare_p:vNv

\dim_compare:nNnTF

\dim_compare:vNnTF

\dim_compare:nNvTF

\dim_compare:vNvTF

```

21679 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
21680 {
21681   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
21682   \prg_return_true: \else: \prg_return_false: \fi:

```

```

21683 }
21684 \cs_new:Npn \dim_compare_p:vNn #1#2#3
21685 { \dim_compare_p:nNn { \use:c {#1} } #2 {#3} }
21686 \cs_new:Npn \dim_compare_p:nNv #1#2#3
21687 { \dim_compare_p:nNn {#1} #2 { \use:c {#3} } }
21688 \cs_new:Npn \dim_compare_p:vNv #1#2#3
21689 { \dim_compare_p:nNn { \use:c {#1} } #2 { \use:c {#3} } }
21690 \clist_map_inline:nn { T , F , TF }
21691 {
21692   \cs_new:cpe { dim_compare:vNn #1 } ##1##2##3
21693   {
21694     \exp_not:c { dim_compare:nNn #1 }
21695     { \exp_not:N \use:c {##1} } ##2 {##3}
21696   }
21697   \cs_new:cpe { dim_compare:nNv #1 } ##1##2##3
21698   {
21699     \exp_not:c { dim_compare:nNn #1 }
21700     {##1} ##2 { \exp_not:N \use:c {##3} }
21701   }
21702   \cs_new:cpe { dim_compare:vNv #1 } ##1##2##3
21703   {
21704     \exp_not:c { dim_compare:nNn #1 }
21705     { \exp_not:N \use:c {##1} } ##2 { \exp_not:N \use:c {##3} }
21706   }
21707 }

```

(End of definition for `\dim_compare:nNnTF`. This function is documented on page 231.)

<pre> \dim_compare_p:n \dim_compare:nTF __dim_compare:w __dim_compare:wNN __dim_compare:=:w __dim_compare!:w __dim_compare<:w __dim_compare>:w __dim_compare_error: </pre>	This code is adapted from the <code>\int_compare:nTF</code> function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing <code>__dim_compare_error:</code>. Just like for integers, the looping auxiliary <code>__dim_compare:wNN</code> closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with <code>pt</code> (with category other). Thus we do not need specific auxiliaries for the three “simple” relations <code><</code>, <code>=</code>, and <code>></code>. <pre> 21708 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF } 21709 { 21710 \exp_after:wN __dim_compare:w 21711 \dim_use:N __dim_eval:w #1 __dim_compare_error: 21712 } 21713 \cs_new:Npn __dim_compare:w #1 __dim_compare_error: 21714 { 21715 \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w 21716 __dim_compare:wNN #1 ? { = __dim_compare_end:w \else: } \s__dim_stop 21717 } 21718 \exp_args:Nno \use:nn 21719 { \cs_new:Npn __dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 } 21720 { 21721 \if_meaning:w = #3 21722 \use:c { __dim_compare_#2:w } 21723 \fi: 21724 #1 pt \exp_stop_f: 21725 \prg_return_false: 21726 \exp_after:wN __dim_use_none_delimit_by_s_stop:w </pre>

```

21727 \fi:
21728 \reverse_if:N \if_dim:w #1 pt #2
21729 \exp_after:wN \__dim_compare:wNN
21730 \dim_use:N \__dim_eval:w #3
21731 }
21732 \cs_new:cpn { \__dim_compare_ ! :w }
21733 #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
21734 \cs_new:cpn { \__dim_compare_ = :w }
21735 #1 \__dim_eval:w = { #1 \__dim_eval:w }
21736 \cs_new:cpn { \__dim_compare_ < :w }
21737 #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
21738 \cs_new:cpn { \__dim_compare_ > :w }
21739 #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
21740 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
21741 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
21742 \cs_new_protected:Npn \__dim_compare_error:
21743 {
21744 \if_int_compare:w \c_zero_int \c_zero_int \fi:
21745 =
21746 \__dim_compare_error:
21747 }

```

(End of definition for \dim_compare:nTF and others. This function is documented on page 232.)

<pre> \dim_case:nn \dim_case:nnTF __dim_case:nnTF __dim_case:nw __dim_case_end:nw </pre>	For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for \str_case:nnTF as described in l3basics. <pre> 21748 \cs_new:Npn \dim_case:nnTF #1 21749 { 21750 \exp:w 21751 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } 21752 } 21753 \cs_new:Npn \dim_case:nnT #1#2#3 21754 { 21755 \exp:w 21756 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { } 21757 } 21758 \cs_new:Npn \dim_case:nnF #1#2 21759 { 21760 \exp:w 21761 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { } 21762 } 21763 \cs_new:Npn \dim_case:nn #1#2 21764 { 21765 \exp:w 21766 \exp_args:Nf __dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { } 21767 } 21768 \cs_new:Npn __dim_case:nnTF #1#2#3#4 21769 { __dim_case:nw {#1} #2 {#1} { } \s__dim_mark {#3} \s__dim_mark {#4} \s__dim_stop } 21770 \cs_new:Npn __dim_case:nw #1#2#3 21771 { 21772 \dim_compare:nNnTF {#1} = {#2} 21773 { __dim_case_end:nw {#3} } 21774 { __dim_case:nw {#1} } 21775 } </pre>

```

21776 \cs_new:Npn \__dim_case_end:nw #1#2#3 \s__dim_mark #4#5 \s__dim_stop
21777 { \exp_end: #1 #4 }

```

(End of definition for `\dim_case:nnTF` and others. This function is documented on page 233.)

69.7 Dimension expression loops

`\dim_while_do:nn` `\dim_until_do:nn` `\dim_do_while:nn` `\dim_do_until:nn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

21778 \cs_new:Npn \dim_while_do:nn #1#2
21779 {
21780   \dim_compare:nT {#1}
21781   {
21782     #2
21783     \dim_while_do:nn {#1} {#2}
21784   }
21785 }
21786 \cs_new:Npn \dim_until_do:nn #1#2
21787 {
21788   \dim_compare:nF {#1}
21789   {
21790     #2
21791     \dim_until_do:nn {#1} {#2}
21792   }
21793 }
21794 \cs_new:Npn \dim_do_while:nn #1#2
21795 {
21796   #2
21797   \dim_compare:nT {#1}
21798   { \dim_do_while:nn {#1} {#2} }
21799 }
21800 \cs_new:Npn \dim_do_until:nn #1#2
21801 {
21802   #2
21803   \dim_compare:nF {#1}
21804   { \dim_do_until:nn {#1} {#2} }
21805 }

```

(End of definition for `\dim_while_do:nn` and others. These functions are documented on page 234.)

`\dim_while_do:nNnn` `\dim_until_do:nNnn` `\dim_do_while:nNnn` `\dim_do_until:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

21806 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
21807 {
21808   \dim_compare:nNnT {#1} #2 {#3}
21809   {
21810     #4
21811     \dim_while_do:nNnn {#1} #2 {#3} {#4}
21812   }
21813 }
21814 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
21815 {
21816   \dim_compare:nNnF {#1} #2 {#3}

```

```

21817 {
21818     #4
21819     \dim_until_do:nNnn {#1} #2 {#3} {#4}
21820 }
21821 }
21822 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
21823 {
21824     #4
21825     \dim_compare:nNnT {#1} #2 {#3}
21826     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
21827 }
21828 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
21829 {
21830     #4
21831     \dim_compare:nNnF {#1} #2 {#3}
21832     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
21833 }

```

(End of definition for `\dim_while_do:nNnn` and others. These functions are documented on page 234.)

69.8 Dimension step functions

```

\dim_step_function:nnnN
  \__dim_step:wwwN
  \__dim_step:NnnnN

```

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

21834 \cs_new:Npn \dim_step_function:nnnN #1#2#3
21835 {
21836     \exp_after:wN \__dim_step:wwwN
21837     \tex_the:D \__dim_eval:w #1 \exp_after:wN \__dim_sep:
21838     \tex_the:D \__dim_eval:w #2 \exp_after:wN \__dim_sep:
21839     \tex_the:D \__dim_eval:w #3 \__dim_sep:
21840 }
21841 \cs_new:Npn \__dim_step:wwwN #1 \__dim_sep: #2 \__dim_sep: #3 \__dim_sep: #4
21842 {
21843     \dim_compare:nNnTF {#2} > \c_zero_dim
21844     { \__dim_step:NnnnN > }
21845     {
21846         \dim_compare:nNnTF {#2} = \c_zero_dim
21847         {
21848             \msg_expandable_error:nnn { kernel } { zero-step } {#4}
21849             \use_none:nnnn
21850         }
21851         { \__dim_step:NnnnN < }
21852     }
21853     {#1} {#2} {#3} #4
21854 }
21855 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
21856 {
21857     \dim_compare:nNnF {#2} #1 {#4}
21858     {
21859         #5 {#2}

```



```

21860         \exp_args:NNf \__dim_step:NnnnN
21861         #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
21862     }
21863 }

```

(End of definition for `\dim_step_function:nnnN`, `\__dim_step:wwwN`, and `\__dim_step:NnnnN`. This function is documented on page 234.)

`\dim_step_inline:nnnn`
`\dim_step_variable:nnnNn`
`\__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

21864 \cs_new_protected:Npn \dim_step_inline:nnnn
21865 {
21866     \int_gincr:N \g__kernel_prg_map_int
21867     \exp_args:NNc \__dim_step:NNnnnn
21868     \cs_gset_protected:Npn
21869     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
21870 }
21871 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
21872 {
21873     \int_gincr:N \g__kernel_prg_map_int
21874     \exp_args:NNc \__dim_step:NNnnnn
21875     \cs_gset_protected:Npe
21876     { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
21877     {#1}{#2}{#3}
21878     {
21879         \tl_set:Nn \exp_not:N #4 {##1}
21880         \exp_not:n {#5}
21881     }
21882 }
21883 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
21884 {
21885     #1 #2 ##1 {#6}
21886     \dim_step_function:nnnN {#3} {#4} {#5} #2
21887     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
21888 }

```

(End of definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `\__dim_step:NNnnnn`. These functions are documented on page 235.)

69.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

21889 \cs_new:Npn \dim_eval:n #1
21890 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End of definition for `\dim_eval:n`. This function is documented on page 235.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since `0.2pt`

and `Opt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

```

21891 \cs_new:Npn \dim_sign:n #1
21892 {
21893   \int_value:w \exp_after:wN \__dim_sign:Nw
21894   \dim_use:N \__dim_eval:w #1 \__dim_eval_end: \__dim_sep:
21895   \exp_stop_f:
21896 }
21897 \cs_new:Npn \__dim_sign:Nw #1#2 \__dim_sep:
21898 {
21899   \if_dim:w #1#2 > \c_zero_dim
21900     1
21901   \else:
21902     \if_meaning:w - #1
21903       -1
21904     \else:
21905       0
21906     \fi:
21907   \fi:
21908 }

```

(End of definition for `\dim_sign:n` and `\__dim_sign:Nw`. This function is documented on page 235.)

`\dim_use:N` Accessing a $\langle dim \rangle$. We hand-code the `c` variant for some speed gain.

```

\dim_use:c
21909 \cs_new_eq:NN \dim_use:N \tex_the:D
21910 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End of definition for `\dim_use:N`. This function is documented on page 235.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```

\__dim_to_decimal:w
21911 \cs_new:Npn \dim_to_decimal:n #1
21912 {
21913   \exp_after:wN
21914   \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
21915 }
21916 \use:e
21917 {
21918   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
21919     #1 . #2 \tl_to_str:n { pt }
21920 }
21921 {
21922   \int_compare:nNnTF {#2} > \c_zero_int
21923     { #1 . #2 }
21924     { #1 }
21925 }

```

(End of definition for `\dim_to_decimal:n` and `\__dim_to_decimal:w`. This function is documented on page 235.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End of definition for `\dim_to_fp:n`. This function is documented on page 237.)

69.10 Conversion of dim to other units

The conversion from `pt` or `sp` to other units is complicated by the fact that $\text{\TeX}$ ’s conversion to `sp` involves rounding and hard-coded ratios. In order to give re-entrant outcomes, we therefore need to do quite a bit of work: see <https://github.com/latex3/latex3/issues/954> for detailed discussion. After dealing with the trivial case, we therefore have some work to do. The code to do this is contributed by Ruixi Zhang.

`\dim_to_decimal_in_sp:n` The one easy case: the only requirement here is that we avoid an overflow.

```
21926 \cs_new:Npn \dim_to_decimal_in_sp:n #1
21927 { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(End of definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 237.)

`\dim_to_decimal_in_bp:n` We first set up a helper macro `\__dim_tmp:w` which takes two arguments. The first
`\dim_to_decimal_in_cc:n` argument is one of the following engine-defined units: `in`, `pc`, `cm`, `mm`, `bp`, `dd`, `cc`, `nd`,
`\dim_to_decimal_in_cm:n` and `nc`. The second argument is $\frac{1}{2}\delta^{-1}$ in reduced fraction, where $\delta > 1$ is the engine-
`\dim_to_decimal_in_dd:n` defined conversion factor for each unit. Note that δ must be strictly larger than 1 for the
`\dim_to_decimal_in_in:n` following algorithm to work.

`\dim_to_decimal_in_mm:n` Here is how the algorithm works: Suppose that a user inputs a non-negative di-
`\dim_to_decimal_in_pc:n` mension in a unit that has conversion factor $\delta > 1$. Then this dimension is internally
`\__dim_to_decimal_aux:w` represented as $X \text{ sp}$, where $X = \lfloor N\delta \rfloor$ for some integer $N \geq 0$. We then seek a formula
to express this N using X . The `\dim_to_decimal_in_<unit>:n` functions shall return
the number $N/2^{16}$ in decimal. This way, we guarantee the returned decimal followed by
the original unit will parse to exactly $X \text{ sp}$.

So how do we get N from X ? Well, since $X = \lfloor N\delta \rfloor$, we have $X \leq N\delta < X + 1$ and $X\delta^{-1} \leq N < (X + 1)\delta^{-1}$. Let’s focus on the midpoint of this bounding interval for N . The midpoint is $(X + \frac{1}{2})\delta^{-1}$. The fact $\delta > 1$ implies that the bounding interval is shorter than 1 in length. Thus, (1) midpoint + $\frac{1}{2} > N$ and (2) midpoint + $\frac{1}{2} < N + 1$. In other words, $N = \lfloor \text{midpoint} + \frac{1}{2} \rfloor$. As long as we can rewrite the midpoint as the result of a “scaling operation” of $\varepsilon\text{-TeX}$, the $\lfloor \dots + \frac{1}{2} \rfloor$ part will follow naturally. Indeed we can: midpoint = $(2X + 1) \times (\frac{1}{2}\delta^{-1})$.

Addendum: If $\delta \geq 2$, then the bounding interval for N is at most $\frac{1}{2}$ wide in length. In this case, the leftpoint $X\delta^{-1}$ suffices as $N = \lfloor X\delta^{-1} + \frac{1}{2} \rfloor$. Six out of the nine units listed above can be handled in this way, which is much simpler than using midpoint. But three remaining units have $1 < \delta < 2$; they are `bp` ($\delta = 7227/7200$), `nd` ($\delta = 685/642$), and `dd` ($\delta = 1238/1157$), and these three must be handled using midpoint. For consistency, we shall use the midpoint approach for all nine units.

```
21928 \group_begin:
21929   \cs_set_protected:Npn \__dim_tmp:w #1#2
21930   {
21931     \cs_new:cpn { dim_to_decimal_in_ #1 :n } ##1
21932     {
21933       \exp_after:wN \__dim_to_decimal_aux:w
21934       \int_value:w \__dim_eval:w ##1 \__dim_eval_end: \__dim_sep: #2 \__dim_sep:
21935     }
21936   }
```

Conversions to other units are now coded. Consult the pdf $\text{\TeX}$ source for each conversion factor δ . Each factor $\frac{1}{2}\delta^{-1}$ is hand-coded for accuracy (and speed). As the units `nc` and `nd` are not supported by XeTeX or (u)p $\text{\TeX}$, they are not included here.

```
21937   \__dim_tmp:w { in } { 50 / 7227 } % delta = 7227/100
```

```

21938 \__dim_tmp:w { pc } { 1 / 24 } % delta = 12/1
21939 \__dim_tmp:w { cm } { 127 / 7227 } % delta = 7227/254
21940 \__dim_tmp:w { mm } { 1270 / 7227 } % delta = 7227/2540
21941 \__dim_tmp:w { bp } { 400 / 803 } % delta = 7227/7200
21942 \__dim_tmp:w { dd } { 1157 / 2476 } % delta = 1238/1157
21943 \__dim_tmp:w { cc } { 1157 / 29712 } % delta = 14856/1157
21944 \group_end:

```

The tokens after `\__dim_to_decimal_aux:w` shall have the following form: `<number>\__dim_sep:<half of delta inverse>\__dim_sep:`, where `<number>` represents the input dimension in `sp` unit. If `<number>` is positive, then `#1` is its leading digit and `#2` (possibly empty) is all the remaining digits; If `<number>` is zero, then `#1` is 0_{12} and `#2` is empty; If `<number>` is negative, then `#1` is its sign $-_{12}$ and `#2` is all its digits. In all three cases, `#1#2` is the original `<number>`. We can use `#1` to decide whether to use the -1 formula or the $+1$ formula.

```

21945 \cs_new:Npn \__dim_to_decimal_aux:w #1#2 \__dim_sep: #3 \__dim_sep:
21946 {
21947   \dim_to_decimal:n
21948   {

```

We need different formulae depending on whether the user input dimension is negative or not. For negative dimension (internally represented as X sp), the formula is $(2X - 1) \times (\frac{1}{2}\delta^{-1})$. For non-negative dimension, the formula is $(2X + 1) \times (\frac{1}{2}\delta^{-1})$. The intermediate step doubles the dimension X . To avoid overflow, we must invoke `\int_eval:n`.

```

21949   \int_eval:n
21950   { ( 2 * #1#2 \if:w #1 - - \else: + \fi: 1 ) * #3 }

```

Now we append `sp` to finish the dimension specification.

```

21951   sp
21952 }
21953 }

```

(End of definition for `\dim_to_decimal_in_bp:n` and others. These functions are documented on page 236.)

`\dim_to_decimal_in_unit:nn`

```

21954 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
21955 {
21956   \exp_after:wN \__dim_chk_unit:w
21957   \int_value:w \__dim_eval:w #2 \__dim_eval_end: \__dim_sep: {#1}
21958 }

```

(End of definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 237.)

`\__dim_chk_unit:w` The tokens after `\__dim_chk_unit:w` shall have the following form: `<number2>\__dim_sep:{<dimexpr1>}`, where `<number2>` represents `<dimexpr2>` in `sp` unit. If `#1` is 0_{12} , the “unit” `<dimexpr2>` must also be zero. So we throw out a “division by zero” error message at this point. Otherwise, if `#1` is $-_{12}$, we shall negate both `<dimexpr1>` and `<dimexpr2>` for later procedures.

```

21959 \cs_new:Npn \__dim_chk_unit:w #1#2 \__dim_sep: #3
21960 {
21961   \token_if_eq_charcode:NNTF #1 0
21962   { \msg_expandable_error:nn { dim } { zero-unit } }
21963   {
21964     \exp_after:wN \__dim_branch_unit:w

```

```

21965         \int_value:w \if:w #1 - - \fi: \_dim_eval:w #3 \exp_after:wN \_dim_sep:
21966         \int_value:w \if:w #1 - - \fi: #1#2 \_dim_sep:
21967     }
21968 }

```

(End of definition for _dim_chk_unit:w.)

_dim_branch_unit:w The tokens after _dim_branch_unit:w shall have the following form: <number1>_dim_sep:<number2>_dim_sep:, where <number1> represents <dimexpr1> in sp unit (whose sign is taken care of) and <number2> represents the absolute value of <dimexpr2> in sp unit (which is strictly positive).

As explained, the formulae $(2X \pm 1) \times (\frac{1}{2}\delta^{-1})$ work if and only if $\delta = \text{<number2>}/65536 > 1$. This corresponds to <dimexpr2> strictly larger than 1pt in absolute value. In this case, we simply call _dim_to_decimal_aux:w and supply $\frac{1}{2}\delta^{-1} = 32768/\text{<number2>}$ as <half of delta inverse>.

Otherwise if <number2> = 65536, then <dimexpr2> is 1pt in absolute value and we call _dim_to_decimal:n directly.

Otherwise $0 < \text{<number2>} < 65536$ and we shall proceed differently.

For unit less than 1pt, write $n = \text{<number2>}$, then $\delta = n/65536 < 1$. The midpoint formulae are not optimal. Let's go back to the inequalities $X\delta^{-1} \leq N < (X+1)\delta^{-1}$. Since now $\delta < 1$, the bounding interval is wider than 1 in length. Consider the ceiling integer $M = \lceil X\delta^{-1} \rceil$, then $X\delta^{-1} \leq M < (X+1)\delta^{-1}$, or equivalently $X \leq M\delta < X+1$, and thus $\lfloor M\delta \rfloor = X$. The key point here is that we *don't* need to solve for N ; in fact, any integer that can reproduce X (such as M) is good enough. So the algorithm goes like this: (1) Compute rounding of $X\delta^{-1}$, i.e., $M' = \lfloor X\delta^{-1} + \frac{1}{2} \rfloor$; this M' could be either M or $M-1$. (2) Check if $\lfloor M'\delta \rfloor = X$, i.e., whether our candidate M' can reproduce X . If so, then this M' is good enough; if not, then we add one to M' .

But when $0 < n < 65536$, we cannot delay the problem of overflow any more. For $X\delta^{-1} = X \times 65536/n$, where X can go up to $2^{30} - 1$ and n can be as small as 1, the result is well over $2^{31} - 1$ (largest integer allowed within \numexpr). For example, _dim_to_decimal_in_unit:nn { \maxdimen } { 1sp }. Here, all inputs are legal, so we should be able to output 1073741823 *without* causing arithmetic overflow.

As a workaround, let's write $X = qn + r$ with some $q \geq 0$ and $0 \leq r < n$. Then $X\delta^{-1} = 65536q + 65536r/n$, and so $M' = 65536q + \lfloor 65536r/n + \frac{1}{2} \rfloor = 65536q + R'$. Computing R' will never overflow. If this R' can reproduce r , then it is good enough; otherwise we add one to R' . In the end, we shall output $q + R'/65536$ in decimal.

Note: $q = \lfloor X/n \rfloor = \lfloor \frac{2X-n}{2n} + \frac{1}{2} \rfloor$ represents the "integer" part, while $0 \leq R' \leq 65536$ represents the "fractional" part. (Can $R' = 65536$ really happen? Didn't investigate.)

```

21969 \cs_new:Npn \_dim_branch_unit:w #1 \_dim_sep: #2 \_dim_sep:
21970 {
21971     \int_compare:nNnTF {#2} > { 65536 }
21972     { \_dim_to_decimal_aux:w #1 \_dim_sep: 32768 / #2 \_dim_sep: }
21973     {
21974         \int_compare:nNnTF {#2} = { 65536 }
21975         { \dim_to_decimal:n { #1sp } }
21976         { \_dim_get_quotient:w #1 \_dim_sep: #2 \_dim_sep: }
21977     }
21978 }

```

(End of definition for _dim_branch_unit:w.)

`\__dim_get_quotient:w` We wish to get the quotient q via rounding of $\frac{2X-n}{2n}$. When $0 \leq X < n/2$, we have $\frac{2X-n}{2n} < 0$. So, strictly speaking, `\numexpr` performs its rounding as $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil$, not exactly what we want. However, lucky for us, only $X = 0$ makes $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = -1 \neq 0$ (we want 0); all other $0 < X < n/2$ make $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = 0 = q$. Thus, let's filter out $X = 0$ early. If $X \neq 0$, we extract its sign and leave the sign to the back. The sign does not participate in any calculations (also the code works with positive integers only). The sign is used at the last stages when we parse the decimal output.

After `\__dim_get_quotient:w` has done its job, either we have the decimal 0, or we have `\__dim_get_remainder:w` followed by $q\backslash\_dim\_sep:n\backslash\_dim\_sep:<sign\ of\ X>\backslash\_dim\_sep:.$

```

21979 \cs_new:Npn \__dim_get_quotient:w #1#2 \__dim_sep: #3 \__dim_sep:
21980 {
21981   \token_if_eq_charcode:NNTF #1 0
21982   { 0 }
21983   {
21984     \token_if_eq_charcode:NNTF #1 -
21985     {
21986       \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
21987       \int_eval:n { ( 2 * #2 - #3 ) / ( 2 * #3 ) } \__dim_sep:
21988       #2 \__dim_sep: #3 \__dim_sep: - \__dim_sep:
21989     }
21990     {
21991       \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
21992       \int_eval:n { ( 2 * #1#2 - #3 ) / ( 2 * #3 ) } \__dim_sep:
21993       #1#2 \__dim_sep: #3 \__dim_sep: \__dim_sep:
21994     }
21995   }
21996 }

```

(End of definition for `\__dim_get_quotient:w`.)

`\__dim_get_remainder:w` `\__dim_get_remainder:w` does not need to read the sign. After finding the remainder r , the number $|X|$ is no longer needed. We should then have `\__dim_convert_remainder:w` followed by $r\backslash\_dim\_sep:n\backslash\_dim\_sep:q\backslash\_dim\_sep:<sign\ of\ X>\backslash\_dim\_sep:.$

```

21997 \cs_new:Npn \__dim_get_remainder:w #1 \__dim_sep: #2 \__dim_sep: #3 \__dim_sep:
21998 {
21999   \exp_after:wN \exp_after:wN \exp_after:wN \__dim_convert_remainder:w
22000   \int_eval:n { #2 - #1 * #3 } \__dim_sep:
22001   #3 \__dim_sep: #1 \__dim_sep:
22002 }

```

(End of definition for `\__dim_get_remainder:w`.)

`\__dim_convert_remainder:w` This is trivial. We compute $R' = \lfloor 65536r/n + \frac{1}{2} \rfloor$, then leave `\__dim_test_candidate:w` followed by $R'\backslash\_dim\_sep:r\backslash\_dim\_sep:n\backslash\_dim\_sep:q\backslash\_dim\_sep:<sign\ of\ X>\backslash\_dim\_sep:.$

```

22003 \cs_new:Npn \__dim_convert_remainder:w #1 \__dim_sep: #2 \__dim_sep:
22004 {
22005   \exp_after:wN \exp_after:wN \exp_after:wN \__dim_test_candidate:w
22006   \int_eval:n { #1 * 65536 / #2 } \__dim_sep:
22007   #1 \__dim_sep: #2 \__dim_sep:
22008 }

```

(End of definition for `\__dim_convert_remainder:w`.)

```

\__dim_test_candidate:w Now the fun part: We take  $R'$ ,  $r$  and  $n$  to test whether  $r = \lfloor R'\delta \rfloor$ . This is done as
a dimension comparison. The left-hand side,  $r$ , is simply r sp. The right-hand side,
 $\lfloor R'\delta \rfloor$ , is exactly <R' as decimal><dimen = n sp>. If the result is true, then we've
found  $R'$ ; otherwise we add one to  $R'$ . After this step,  $r$  and  $n$  are no longer needed.
We should then have \__dim_parse_decimal:w followed by  $R'\backslash\_dim\_sep:q\backslash\_dim\_sep:
sep:<sign\ of\ X>\_dim\_sep:$ .
22009 \cs_new:Npn \__dim_test_candidate:w #1 \__dim_sep: #2 \__dim_sep: #3 \__dim_sep:
22010 {
22011   \dim_compare:nNnTF { #2sp } =
22012     { \dim_to_decimal:n { #1sp } \__dim_eval:w #3sp \__dim_eval_end: }
22013     { \__dim_parse_decimal:w #1 \__dim_sep: }
22014     {
22015       \__dim_parse_decimal:w \int_eval:n { #1 + 1 } \__dim_sep:
22016     }
22017 }

```

(End of definition for `\__dim_test_candidate:w`.)

```

\__dim_parse_decimal:w The Grand Finale: We sum  $q$  and  $R'/65536$  together, and negate the result if necessary.
\__dim_parse_decimal_aux:w These are all done expandably. If  $0 < R'/65536 < 1$ , the integer summation is naturally
terminated at the decimal point. If  $R'/65536 = 0$  (or  $1?$ ), the summation is terminated
at the \__dim_sep:. The auxiliary function \__dim_parse_decimal_aux:w takes care
of both cases.
22018 \cs_new:Npn \__dim_parse_decimal:w #1 \__dim_sep: #2 \__dim_sep: #3 \__dim_sep:
22019 {
22020   \exp_after:wN \__dim_parse_decimal_aux:w
22021   \int_value:w #3 \int_eval:w #2 + \dim_to_decimal:n { #1sp } \__dim_sep:
22022 }
22023 \cs_new:Npn \__dim_parse_decimal_aux:w #1 \__dim_sep: {#1}

```

(End of definition for `\__dim_parse_decimal:w` and `\__dim_parse_decimal_aux:w`.)

69.11 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c 22024 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
22025 \cs_generate_variant:Nn \dim_show:N { c }

```

(End of definition for `\dim_show:N`. This function is documented on page 237.)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```

22026 \cs_new_protected:Npn \dim_show:n
22027 { \__kernel_msg_show_eval:Nn \dim_eval:n }

```

(End of definition for `\dim_show:n`. This function is documented on page 237.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```

\dim_log:c 22028 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\dim_log:n 22029 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
22030 \cs_new_protected:Npn \dim_log:n
22031 { \__kernel_msg_log_eval:Nn \dim_eval:n }

```

(End of definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 237.)

69.12 Constant dimensions

`\c_zero_dim` Constant dimensions.

`\c_max_dim` 22032 `\dim_const:Nn \c_zero_dim { 0 pt }`
 22033 `\dim_const:Nn \c_max_dim { 16383.99999 pt }`

(End of definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 238.)

69.13 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_dim` 22034 `\dim_new:N \l_tmpa_dim`
`\g_tmpa_dim` 22035 `\dim_new:N \l_tmpb_dim`
`\g_tmpb_dim` 22036 `\dim_new:N \g_tmpa_dim`
 22037 `\dim_new:N \g_tmpb_dim`

(End of definition for `\l_tmpa_dim` and others. These variables are documented on page 238.)

69.14 Inserting dimensions into the output

`\dim_horizontal:n` TeX does not treat kerns differently in horizontal and vertical mode. As we want skips and kerns to have similar behavior, we try to get similar results.

`\dim_horizontal:N` 22038 `\cs_new_protected:Npn \dim_horizontal:n #1`
`\dim_horizontal:c` 22039 `{`
`\dim_vertical:n` 22040 `\mode_leave_vertical:`
`\dim_vertical:N` 22041 `\__dim_kern:n {#1}`
`\dim_vertical:c` 22042 `}`
`\__dim_kern:n` 22043 `\cs_set_eq:NN \dim_horizontal:N \dim_horizontal:n`
 22044 `\cs_generate_variant:Nn \dim_horizontal:N { c }`
 22045 `\cs_new_protected:Npn \dim_vertical:n #1`
 22046 `{`
 22047 `\mode_if_vertical:F`
 22048 `{ \par }`
 22049 `\__dim_kern:n {#1}`
 22050 `}`
 22051 `\cs_set_eq:NN \dim_vertical:N \dim_vertical:n`
 22052 `\cs_generate_variant:Nn \dim_vertical:N { c }`
 22053 `\cs_new_protected:Npn \__dim_kern:n #1`
 22054 `{ \tex_kern:D \tex_dimexpr:D #1 \scan_stop: }`

(End of definition for `\dim_horizontal:n` and others. These functions are documented on page 238.)

69.15 Creating and initializing skip variables

22055 `<@@=skip>`

`\s__skip_stop` Internal scan marks.

22056 `\scan_new:N \s__skip_stop`

(End of definition for `\s__skip_stop`.)

\skip_new:N Allocation of a new internal registers.
\skip_new:c

```

22057 \cs_new_protected:Npn \skip_new:N #1
22058 {
22059     \__kernel_chk_if_free_cs:N #1
22060     \cs:w newskip \cs_end: #1
22061 }
22062 \cs_generate_variant:Nn \skip_new:N { c }

```

(End of definition for \skip_new:N. This function is documented on page 238.)

\skip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants. See
\skip_const:cn \dim_const:Nn for why we cannot use \skip_gset:Nn.

```

22063 \cs_new_protected:Npn \skip_const:Nn #1#2
22064 {
22065     \skip_new:N #1
22066     \tex_global:D #1 = \skip_eval:n {#2} \scan_stop:
22067 }
22068 \cs_generate_variant:Nn \skip_const:Nn { c }

```

(End of definition for \skip_const:Nn. This function is documented on page 239.)

\skip_zero:N Reset the register to zero.

\skip_zero:c

```

22069 \cs_new_eq:NN \skip_zero:N \dim_zero:N

```

\skip_gzero:N

```

22070 \cs_new_eq:NN \skip_gzero:N \dim_gzero:N

```

\skip_gzero:c

```

22071 \cs_generate_variant:Nn \skip_zero:N { c }
22072 \cs_generate_variant:Nn \skip_gzero:N { c }

```

(End of definition for \skip_zero:N and \skip_gzero:N. These functions are documented on page 239.)

\skip_zero_new:N Create a register if needed, otherwise clear it.

\skip_zero_new:c

```

22073 \cs_new_protected:Npn \skip_zero_new:N #1

```

\skip_gzero_new:N

```

22074 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }

```

\skip_gzero_new:c

```

22075 \cs_new_protected:Npn \skip_gzero_new:N #1
22076 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
22077 \cs_generate_variant:Nn \skip_zero_new:N { c }
22078 \cs_generate_variant:Nn \skip_gzero_new:N { c }

```

(End of definition for \skip_zero_new:N and \skip_gzero_new:N. These functions are documented on page 239.)

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.

\skip_if_exist_p:c

```

22079 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N

```

\skip_if_exist:NTF

```

22080 { TF , T , F , p }

```

\skip_if_exist:cTF

```

22081 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
22082 { TF , T , F , p }

```

(End of definition for \skip_if_exist:NTF. This function is documented on page 239.)

69.16 Setting skip variables

`\skip_set:Nn` Much the same as for dimensions.

```
\skip_set:cn 22083 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_set:NV 22084 { #1 = \tex_glueexpr:D #2 \scan_stop: }
\skip_set:cV 22085 \cs_new_protected:Npn \skip_gset:Nn #1#2
\skip_gset:Nn 22086 { \tex_global:D #1 = \tex_glueexpr:D #2 \scan_stop: }
\skip_gset:cn 22087 \cs_generate_variant:Nn \skip_set:Nn { NV , c , cV }
\skip_gset:NV 22088 \cs_generate_variant:Nn \skip_gset:Nn { NV , c , cV }
\skip_gset:cV
```

(End of definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 239.)

`\skip_set_eq:NN` All straightforward.

```
\skip_set_eq:cn 22089 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 22090 \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\skip_set_eq:cc 22091 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:Nn 22092 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
```

(End of definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 239.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```
\skip_add:cn 22093 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:Nn 22094 { \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\skip_gadd:cn 22095 \cs_new_protected:Npn \skip_gadd:Nn #1#2
\skip_sub:Nn 22096 { \tex_global:D \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\skip_sub:cn 22097 \cs_generate_variant:Nn \skip_add:Nn { c }
\skip_gsub:Nn 22098 \cs_generate_variant:Nn \skip_gadd:Nn { c }
\skip_gsub:cn 22099 \cs_new_protected:Npn \skip_sub:Nn #1#2
22100 { \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
22101 \cs_new_protected:Npn \skip_gsub:Nn #1#2
22102 { \tex_global:D \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
22103 \cs_generate_variant:Nn \skip_sub:Nn { c }
22104 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End of definition for `\skip_add:Nn` and others. These functions are documented on page 239.)

69.17 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```
22105 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
22106 {
22107   \str_if_eq:eeTF { \skip_eval:n {#1} } { \skip_eval:n {#2} }
22108   { \prg_return_true: }
22109   { \prg_return_false: }
22110 }
```

(End of definition for `\skip_if_eq:nnTF`. This function is documented on page 240.)

`__skip_sep:`

```
22111 \cs_new_eq:NN __skip_sep: __kernel_int_sep:
```

(End of definition for `__skip_sep:.`)

`\skip_if_finite:p:n` With ε -TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

`\skip_if_finite:nTF`

`\__skip_if_finite:wwNw`

```

22112 \cs_set_protected:Npn \__skip_tmp:w #1
22113 {
22114   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
22115   {
22116     \exp_after:wN \__skip_if_finite:wwNw
22117     \skip_use:N \tex_glueexpr:D ##1 \__skip_sep: \prg_return_false:
22118     #1 \__skip_sep: \prg_return_true: \s__skip_stop
22119   }
22120   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 \__skip_sep: ##3 ##4 \s__skip_stop {##3}
22121 }
22122 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }

```

(End of definition for `\skip_if_finite:nTF` and `\__skip_if_finite:wwNw`. This function is documented on page 240.)

69.18 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

22123 \cs_new:Npn \skip_eval:n #1
22124 { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }

```

(End of definition for `\skip_eval:n`. This function is documented on page 240.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

`\skip_use:c`

```

22125 \cs_new_eq:NN \skip_use:N \dim_use:N
22126 \cs_new_eq:NN \skip_use:c \dim_use:c

```

(End of definition for `\skip_use:N`. This function is documented on page 240.)

69.19 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c 22127 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n 22128 \cs_new:Npn \skip_horizontal:n #1
\skip_vertical:N   22129 { \tex_hskip:D \tex_glueexpr:D #1 \scan_stop: }
\skip_vertical:c   22130 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
\skip_vertical:n   22131 \cs_new:Npn \skip_vertical:n #1
22132 { \tex_vskip:D \tex_glueexpr:D #1 \scan_stop: }
22133 \cs_generate_variant:Nn \skip_horizontal:N { c }
22134 \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End of definition for `\skip_horizontal:N` and others. These functions are documented on page 241.)

69.20 Viewing skip variables

`\skip_show:N` Diagnostics.

```
\skip_show:c 22135 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
22136 \cs_generate_variant:Nn \skip_show:N { c }
```

(End of definition for \skip_show:N. This function is documented on page 240.)

`\skip_show:n` Diagnostics. We don't use the T_EX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
22137 \cs_new_protected:Npn \skip_show:n
22138 { \__kernel_msg_show_eval:Nn \skip_eval:n }
```

(End of definition for \skip_show:n. This function is documented on page 240.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```
\skip_log:c 22139 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 22140 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
22141 \cs_new_protected:Npn \skip_log:n
22142 { \__kernel_msg_log_eval:Nn \skip_eval:n }
```

(End of definition for \skip_log:N and \skip_log:n. These functions are documented on page 241.)

69.21 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

```
\c_max_skip 22143 \skip_const:Nn \c_zero_skip { \c_zero_dim }
22144 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(End of definition for \c_zero_skip and \c_max_skip. These functions are documented on page 241.)

69.22 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

```
\l_tmpb_skip 22145 \skip_new:N \l_tmpa_skip
\g_tmpa_skip 22146 \skip_new:N \l_tmpb_skip
\g_tmpb_skip 22147 \skip_new:N \g_tmpa_skip
22148 \skip_new:N \g_tmpb_skip
```

(End of definition for \l_tmpa_skip and others. These variables are documented on page 241.)

69.23 Creating and initializing muskip variables

`\muskip_new:N` And then we add muskips.

```
\muskip_new:c 22149 \cs_new_protected:Npn \muskip_new:N #1
22150 {
22151 \__kernel_chk_if_free_cs:N #1
22152 \cs:w newmuskip \cs_end: #1
22153 }
22154 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End of definition for \muskip_new:N. This function is documented on page 242.)

\muskip_const:Nn See \skip_const:Nn.
\muskip_const:cn 22155 \cs_new_protected:Npn \muskip_const:Nn #1#2
22156 {
22157 \muskip_new:N #1
22158 \tex_global:D #1 = \muskip_eval:n {#2} \scan_stop:
22159 }
22160 \cs_generate_variant:Nn \muskip_const:Nn { c }
(End of definition for \muskip_const:Nn. This function is documented on page 242.)

\muskip_zero:N Reset the register to zero.
\muskip_zero:c 22161 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:N 22162 { #1 = \c_zero_muskip }
\muskip_gzero:c 22163 \cs_new_protected:Npn \muskip_gzero:N #1
22164 { \tex_global:D #1 = \c_zero_muskip }
22165 \cs_generate_variant:Nn \muskip_zero:N { c }
22166 \cs_generate_variant:Nn \muskip_gzero:N { c }
(End of definition for \muskip_zero:N and \muskip_gzero:N. These functions are documented on page 242.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.
\muskip_zero_new:c 22167 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 22168 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 22169 \cs_new_protected:Npn \muskip_gzero_new:N #1
22170 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
22171 \cs_generate_variant:Nn \muskip_zero_new:N { c }
22172 \cs_generate_variant:Nn \muskip_gzero_new:N { c }
(End of definition for \muskip_zero_new:N and \muskip_gzero_new:N. These functions are documented on page 242.)

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.
\muskip_if_exist_p:c 22173 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:N~~TF~~ 22174 { TF , T , F , p }
\muskip_if_exist:c~~TF~~ 22175 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
22176 { TF , T , F , p }
(End of definition for \muskip_if_exist:N~~TF~~. This function is documented on page 242.)

69.24 Setting muskip variables

\muskip_set:Nn This should be pretty familiar.
\muskip_set:cn 22177 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_set:NV 22178 { #1 = \tex_muexpr:D #2 \scan_stop: }
\muskip_set:cV 22179 \cs_new_protected:Npn \muskip_gset:Nn #1#2
\muskip_gset:Nn 22180 { \tex_global:D #1 = \tex_muexpr:D #2 \scan_stop: }
\muskip_gset:cn 22181 \cs_generate_variant:Nn \muskip_set:Nn { NV , c , cV }
\muskip_gset:NV 22182 \cs_generate_variant:Nn \muskip_gset:Nn { NV , c , cV }
\muskip_gset:cV *(End of definition for \muskip_set:Nn and \muskip_gset:Nn. These functions are documented on page 242.)*

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cN 22183 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:Nc 22184 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_set_eq:cc 22185 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\muskip_gset_eq:NN 22186 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

```

(End of definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 242.)

`\muskip_add:Nn` Using by here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cN 22187 \cs_new_protected:Npn \muskip_add:Nn #1#2
\muskip_gadd:Nn 22188 { \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_gadd:cN 22189 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
\muskip_sub:Nn 22190 { \tex_global:D \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\muskip_sub:cN 22191 \cs_generate_variant:Nn \muskip_add:Nn { c }
\muskip_gsub:Nn 22192 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
\muskip_gsub:cN 22193 \cs_new_protected:Npn \muskip_sub:Nn #1#2
22194 { \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
22195 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
22196 { \tex_global:D \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
22197 \cs_generate_variant:Nn \muskip_sub:Nn { c }
22198 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End of definition for `\muskip_add:Nn` and others. These functions are documented on page 242.)

69.25 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

22199 \cs_new:Npn \muskip_eval:n #1
22200 { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }

```

(End of definition for `\muskip_eval:n`. This function is documented on page 243.)

`\muskip_use:N` Accessing a `\muskip`.

```

\muskip_use:c 22201 \cs_new_eq:NN \muskip_use:N \dim_use:N
22202 \cs_new_eq:NN \muskip_use:c \dim_use:c

```

(End of definition for `\muskip_use:N`. This function is documented on page 243.)

69.26 Viewing muskip variables

`\muskip_show:N` Diagnostics.

```

\muskip_show:c 22203 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
22204 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End of definition for `\muskip_show:N`. This function is documented on page 243.)

`\muskip_show:n` Diagnostics. We don't use the $\TeX$ primitive `\showthe` to show muskip expressions: this gives a more unified output.

```

22205 \cs_new_protected:Npn \muskip_show:n
22206 { \__kernel_msg_show_eval:Nn \muskip_eval:n }

```

(End of definition for `\muskip_show:n`. This function is documented on page 243.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.

`\muskip_log:c` 22207 `\cs_new_eq:NN \muskip_log:N \__kernel_register_log:N`

`\muskip_log:n` 22208 `\cs_new_eq:NN \muskip_log:c \__kernel_register_log:c`

22209 `\cs_new_protected:Npn \muskip_log:n`

22210 `{ \__kernel_msg_log_eval:Nn \muskip_eval:n }`

(End of definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 243.)

69.27 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.

`\c_max_muskip` 22211 `\muskip_const:Nn \c_zero_muskip { 0 mu }`

22212 `\muskip_const:Nn \c_max_muskip { 16383.99999 mu }`

(End of definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 243.)

69.28 Scratch muskips

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_muskip` 22213 `\muskip_new:N \l_tmpa_muskip`

`\g_tmpa_muskip` 22214 `\muskip_new:N \l_tmpb_muskip`

`\g_tmpb_muskip` 22215 `\muskip_new:N \g_tmpa_muskip`

22216 `\muskip_new:N \g_tmpb_muskip`

(End of definition for `\l_tmpa_muskip` and others. These variables are documented on page 244.)

22217 `\code`

Chapter 70

l3keys implementation

22218 $\langle *code \rangle$

70.1 Low-level interface

The low-level key parser's implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional “safety” requirements and allows to process the parsed list of key–value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimize speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

22219 $\langle @@=keyval \rangle$

```
\s__keyval_nil
\s__keyval_mark
\s__keyval_stop
\s__keyval_tail
22220 \scan_new:N \s__keyval_nil
22221 \scan_new:N \s__keyval_mark
22222 \scan_new:N \s__keyval_stop
22223 \scan_new:N \s__keyval_tail
```

(End of definition for `\s__keyval_nil` and others.)

`\l__kernel_keyval_allow_blank_keys_bool`

The general behavior of the `l3keys` module is to throw an error on blank key names. However to support the usage of `\keyval_parse:nnn` in the `l3prop` module we allow this error to be switched off temporarily and just ignore blank names.

22224 $\backslash bool\_new:N \l__kernel\_keyval\_allow\_blank\_keys\_bool$

(End of definition for `\l__kernel_keyval_allow_blank_keys_bool`.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro `#1` will be the active comma and `#2` will be the active equals sign.

```
22225 \group_begin:
22226   \cs_set_protected:Npn \__keyval_tmp:nn #1#2
22227   {
```

```
\keyval_parse:nnn
\keyval_parse:nnV
\keyval_parse:nnv
\keyval_parse:NNn
\keyval_parse:NNV
\keyval_parse:NNv
```

The main function starts the first of two loops. The outer loop splits the key–value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.


```

22228 \cs_new:Npn \keyval_parse:nnn ##1 ##2 ##3
22229 {
22230   \__kernel_exp_not:w \tex_expanded:D
22231   {
22232     {
22233       \__keyval_loop_active:nnw {##1} {##2}
22234       \s__keyval_mark ##3 #1 \s__keyval_tail #1
22235     }
22236   }
22237 }
22238 \cs_new_eq:NN \keyval_parse:NNn \keyval_parse:nnn

```

(End of definition for \keyval_parse:nnn and \keyval_parse:NNn. These functions are documented on page 259.)

__keyval_loop_active:nnw

First a fast test for the end of the loop is done, it'll gobble everything up to a \s__keyval_tail. The loop ending macro will gobble everything to the last comma in this definition. If the end isn't reached yet, start the second loop splitting at other commas, the next iteration of this first loop will be inserted by the end of __keyval_loop_other:nnw.

```

22239 \cs_new:Npn \__keyval_loop_active:nnw ##1 ##2 ##3 #1
22240 {
22241   \__keyval_if_recursion_tail:w ##3
22242   \__keyval_end_loop_active:w \s__keyval_tail
22243   \__keyval_loop_other:nnw {##1} {##2} ##3 , \s__keyval_tail ,
22244 }

```

(End of definition for __keyval_loop_active:nnw.)

__keyval_split_other:w
 __keyval_split_active:w

These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following \s__keyval_mark that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```

22245 \cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3
22246 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
22247 \cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3
22248 { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }

```

(End of definition for __keyval_split_other:w and __keyval_split_active:w.)

__keyval_loop_other:nnw

The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using __keyval_split_active:w. The \s__keyval_nil prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```

22249 \cs_new:Npn \__keyval_loop_other:nnw ##1 ##2 ##3 ,
22250 {
22251   \__keyval_if_recursion_tail:w ##3
22252   \__keyval_end_loop_other:w \s__keyval_tail
22253   \__keyval_split_active:w ##3 \s__keyval_nil
22254   \s__keyval_mark \__keyval_split_active_auxi:w
22255   #2 \s__keyval_mark \__keyval_clean_up_active:w
22256   {##1} {##2}
22257   \s__keyval_mark
22258 }

```

(End of definition for `\__keyval_loop_other:nnw`.)

`\__keyval_split_active_auxi:w` After `\__keyval_split_active:w` the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. `##1` will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via `\__keyval_misplaced_equal_after_active_error:w`. If none was found we forward the key to `\__keyval_split_active_auxii:w`.

```

22259     \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
22260     {
22261         \__keyval_split_other:w ##1 \s__keyval_nil
22262         \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
22263         = \s__keyval_mark \__keyval_split_active_auxii:w
22264     }

```

`\__keyval_split_active_auxii:w` gets the correct key name with a leading `\s__keyval_mark` as `##1`. It has to sanitize the remainder of the previous test and trims the key name which will be forwarded to `\__keyval_split_active_auxiii:w`.

```

22265     \cs_new:Npn \__keyval_split_active_auxii:w
22266     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
22267     \s__keyval_stop \s__keyval_mark
22268     ##2 \s__keyval_nil #2 \s__keyval_mark \__keyval_clean_up_active:w
22269     { \__keyval_trim:nN {##1} \__keyval_split_active_auxiii:w ##2 \s__keyval_nil }

```

Next we test for a misplaced active equals sign in the value, if none is found `\__keyval_split_active_auxiv:w` will be called.

```

22270     \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
22271     {
22272         \__keyval_split_active:w ##2 \s__keyval_nil
22273         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22274         #2 \s__keyval_mark \__keyval_split_active_auxiv:w
22275         {##1}
22276     }

```

This runs the last test after sanitizing the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```

22277     \cs_new:Npn \__keyval_split_active_auxiv:w
22278     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22279     \s__keyval_stop \s__keyval_mark
22280     {
22281         \__keyval_split_other:w ##1 \s__keyval_nil
22282         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22283         = \s__keyval_mark \__keyval_split_active_auxv:w
22284     }

```

This last macro in this execution branch sanitizes the last test, trims the value and passes it to `\__keyval_pair:nnnn`.

```

22285     \cs_new:Npn \__keyval_split_active_auxv:w
22286     ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22287     \s__keyval_stop \s__keyval_mark
22288     { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

(End of definition for `\__keyval_split_active_auxi:w` and others.)

`\__keyval_clean_up_active:w` The following is the branch taken if the key-value pair doesn't contain an active equals sign. The remainder of that test will be cleaned up by `\__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

22289     \cs_new:Npn \__keyval_clean_up_active:w
22290         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
22291     {
22292         \__keyval_split_other:w ##1 \s__keyval_nil
22293         \s__keyval_mark \__keyval_split_other_auxi:w
22294         = \s__keyval_mark \__keyval_clean_up_other:w
22295     }

```

(End of definition for __keyval_clean_up_active:w.)

`\__keyval_split_other_auxi:w` This is executed if the key-value pair doesn't contain an active equals sign but at least one other. `##1` of `\__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

`\__keyval_split_other_auxii:w`
`\__keyval_split_other_auxiii:w`

```

22296     \cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
22297     { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxii:w }

```

We know that the value doesn't contain misplaced active equals signs but we have to test for others. Also we need to sanitize the previous test, which is done here and not earlier to avoid superfluous argument grabbing.

```

22298     \cs_new:Npn \__keyval_split_other_auxii:w
22299         ##1 ##2 \s__keyval_nil = \s__keyval_mark \__keyval_clean_up_other:w
22300     {
22301         \__keyval_split_other:w ##2 \s__keyval_nil
22302         \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22303         = \s__keyval_mark \__keyval_split_other_auxiii:w
22304         { ##1 }
22305     }

```

`\__keyval_split_other_auxiii:w` sanitizes the test for other equals signs, trims the value and forwards it to `\__keyval_pair:nnnn`.

```

22306     \cs_new:Npn \__keyval_split_other_auxiii:w
22307         ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
22308         \s__keyval_stop \s__keyval_mark
22309         { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

(End of definition for __keyval_split_other_auxi:w, __keyval_split_other_auxii:w, and __keyval_split_other_auxiii:w.)

`\__keyval_clean_up_other:w` `\__keyval_clean_up_other:w` is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to `\__keyval_key:nn`.

```

22310     \cs_new:Npn \__keyval_clean_up_other:w
22311         ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \
22312     {
22313         \__keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
22314         \s__keyval_mark \s__keyval_stop
22315         \__keyval_trim:nN { ##1 } \__keyval_key:nn
22316     }

```

(End of definition for __keyval_clean_up_other:w.)

keyval_misplaced_equal_after_active_error:w
 _keyval_misplaced_equal_in_split_error:w

All these two macros do is gobble the remainder of the current other loop execution and throw an error. Afterwards they have to insert the next loop iteration.

```

22317 \cs_new:Npn \_keyval_misplaced_equal_after_active_error:w
22318 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
22319 = \s__keyval_mark \_keyval_split_active_auxii:w
22320 \s__keyval_mark ##3 \s__keyval_nil
22321 #2 \s__keyval_mark \_keyval_clean_up_active:w
22322 {
22323 \msg_expandable_error:nn
22324 { keyval } { misplaced-equals-sign }
22325 \_keyval_loop_other:nnw
22326 }
22327 \cs_new:Npn \_keyval_misplaced_equal_in_split_error:w
22328 \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
22329 ##3 \s__keyval_mark ##4 ##5
22330 {
22331 \msg_expandable_error:nn
22332 { keyval } { misplaced-equals-sign }
22333 \_keyval_loop_other:nnw
22334 }

```

(End of definition for _keyval_misplaced_equal_after_active_error:w and _keyval_misplaced_equal_in_split_error:w.)

_keyval_end_loop_other:w
 _keyval_end_loop_active:w

All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call. _keyval_end_loop_other:w also has to insert the next iteration of the active loop.

```

22335 \cs_new:Npn \_keyval_end_loop_other:w
22336 \s__keyval_tail
22337 \_keyval_split_active:w
22338 \s__keyval_mark \s__keyval_tail
22339 \s__keyval_nil \s__keyval_mark
22340 \_keyval_split_active_auxi:w
22341 #2 \s__keyval_mark \_keyval_clean_up_active:w
22342 { \_keyval_loop_active:nnw }
22343 \cs_new:Npn \_keyval_end_loop_active:w
22344 \s__keyval_tail
22345 \_keyval_loop_other:nnw ##1 \s__keyval_mark \s__keyval_tail , \s__keyval_tail ,
22346 { }

```

(End of definition for _keyval_end_loop_other:w and _keyval_end_loop_active:w.)

The parsing loops are done, so here ends the definition of _keyval_tmp:nn, which will finally set up the macros.

```

22347 }
22348 \char_set_catcode_active:n { '\, }
22349 \char_set_catcode_active:n { '\= }
22350 \_keyval_tmp:nn , =
22351 \group_end:
22352 \cs_generate_variant:Nn \keyval_parse:NNn { NNv , NNv }
22353 \cs_generate_variant:Nn \keyval_parse:nnn { nnV , nnv }

```

_keyval_pair:nnnn
 _keyval_key:nn

These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the func-

tions passed to `\keyval_parse:nnn` inside of `\exp_not:n` with the correct arguments. Afterwards they insert the next iteration of the other loop.

```

22354 \group_begin:
22355   \cs_set_protected:Npn \__keyval_tmp:nn #1#2
22356   {
22357     \cs_new:Npn \__keyval_pair:nnnn ##1 ##2 ##3 ##4
22358     {
22359       \__keyval_if_blank:w \s__keyval_mark ##2 \s__keyval_nil \s__keyval_stop \__keyval
22360       \s__keyval_mark \s__keyval_stop
22361       #1
22362       \exp_not:n { ##4 {##2} {##1} }
22363       #2
22364       \__keyval_loop_other:nnw {##3} {##4}
22365     }
22366     \cs_new:Npn \__keyval_key:nn ##1 ##2
22367     {
22368       \__keyval_if_blank:w \s__keyval_mark ##1 \s__keyval_nil \s__keyval_stop \__keyval
22369       \s__keyval_mark \s__keyval_stop
22370       #1
22371       \exp_not:n { ##2 {##1} }
22372       #2
22373       \__keyval_loop_other:nnw {##2}
22374     }
22375   }
22376   \__keyval_tmp:nn { } { }
22377 \group_end:

```

(End of definition for `\__keyval_pair:nnnn` and `\__keyval_key:nn`.)

`\__keyval_if_empty:w` All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval_mark`.

```

22378 \cs_new:Npn \__keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
22379 \cs_new:Npn \__keyval_if_blank:w \s__keyval_mark #1 { \__keyval_if_empty:w \s__keyval_mark
22380 \cs_new:Npn \__keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }

```

(End of definition for `\__keyval_if_empty:w`, `\__keyval_if_blank:w`, and `\__keyval_if_recursion_tail:w`.)

`\__keyval_blank_true:w` These macros will be called if the tests above didn't gobble them, they execute the branching.

```

22381 \cs_new:Npn \__keyval_blank_true:w \s__keyval_mark \s__keyval_stop \__keyval_trim:nN #1 \__
22382 { \__keyval_loop_other:nnw }
22383 \cs_new:Npn \__keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop #1 \__keyval_loop_o
22384 {
22385   \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
22386   { #1 }
22387   { \msg_expandable_error:nn { keyval } { blank-key-name } }
22388   \__keyval_loop_other:nnw
22389 }

```

(End of definition for `\__keyval_blank_true:w` and `\__keyval_blank_key_error:w`.)

Two messages for the low level parsing system.

```

22390 \msg_new:nnn { keyval } { misplaced-equals-sign }
22391 { Misplaced~'='~in~key-value-input~\msg_line_context: }
22392 \msg_new:nnn { keyval } { blank-key-name }
22393 { Blank~key~name~in~key-value-input~\msg_line_context: }
22394 \prop_gput:Nnn \g_msg_module_name_prop { keyval } { LaTeX }
22395 \prop_gput:Nnn \g_msg_module_type_prop { keyval } { }

```

`\__keyval_trim:nN` And an adapted version of `\__tl_trim_spaces:nn` which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of `\tl_trim_spaces_apply:nN` is about 10 % of the total time for `\keyval_parse:NNn` with one key and one key–value pair, so I think it's worth it.

```

22396 \group_begin:
22397   \cs_set_protected:Npn \__keyval_tmp:n #1
22398   {
22399     \cs_new:Npn \__keyval_trim:nN ##1
22400     {
22401       \__keyval_trim_auxi:w
22402       ##1
22403       \s__keyval_nil
22404       \s__keyval_mark #1 { }
22405       \s__keyval_mark \__keyval_trim_auxii:w
22406       \__keyval_trim_auxiii:w
22407       #1 \s__keyval_nil
22408       \__keyval_trim_auxiv:w
22409     }
22410     \cs_new:Npn \__keyval_trim_auxi:w ##1 \s__keyval_mark #1 ##2 \s__keyval_mark ##3
22411     {
22412       ##3
22413       \__keyval_trim_auxi:w
22414       \s__keyval_mark
22415       ##2
22416       \s__keyval_mark #1 {##1}
22417     }
22418     \cs_new:Npn \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval_mark \s__keyval_m
22419     {
22420       \__keyval_trim_auxiii:w
22421       ##1
22422     }
22423     \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
22424     {
22425       ##2
22426       ##1 \s__keyval_nil
22427       \__keyval_trim_auxiii:w
22428     }

```

This is the one macro which differs from the original definition.

```

22429   \cs_new:Npn \__keyval_trim_auxiv:w
22430   \s__keyval_mark ##1 \s__keyval_nil
22431   \__keyval_trim_auxiii:w \s__keyval_nil \__keyval_trim_auxiii:w
22432   ##2
22433   { ##2 { ##1 } }
22434 }
22435 \__keyval_tmp:n { ~ }

```

22436 \group_end:

(End of definition for __keyval_trim:nN and others.)

\keyval_map_inline:nnn

Unlike many other mapping functions we don't need to keep a global stack of definitions for our auxiliary macros, instead we can contain their definition in a group and use \keyval_parse:NNn which we fully expand twice before the group is closed. This has the drawback that any breaking of the map happens after the entire list was already parsed, so has a performance penalty in that case. However, these should be rather rare occasions and the simplicity of the code should warrant them. Another drawback of this approach is that the expansion of the internal macros is left in the input stream for every element.

The break point can remain outside of the braces containing the loop, since by the time any user code from the argument to the break statement is executed, the brace group will have been consumed.

```
22437 \cs_new_protected:Npn \keyval_map_inline:nnn #1#2#3
22438 {
22439   \group_begin:
22440   \cs_set:Npn \__keyval_tmp:n ##1 { \exp_not:n {#2} }
22441   \cs_set:Npn \__keyval_tmp:nn ##1##2 { \exp_not:n {#3} }
22442   \exp_args:NNe \exp_last_unbraced:Ne
22443   \group_end:
22444   { \keyval_parse:NNn \__keyval_tmp:n \__keyval_tmp:nn {#1} }
22445   \prg_break_point:Nn \keyval_map_break: { }
22446 }
```

(End of definition for \keyval_map_inline:nnn. This function is documented on page 260.)

\keyval_map_break:

Simple code using the \prg_map_break:Nn mechanism.

\keyval_map_break:n

```
22447 \cs_new:Npn \keyval_map_break: { \prg_map_break:Nn \keyval_map_break: { } }
22448 \cs_new:Npn \keyval_map_break:n { \prg_map_break:Nn \keyval_map_break: }
```

(End of definition for \keyval_map_break: and \keyval_map_break:n. These functions are documented on page 261.)

70.2 Constants and variables

22449 <@@=keys>

Various storage areas for the different data which make up keys.

```
\c__keys_code_root_str
\c__keys_check_root_str
\c__keys_default_root_str
\c__keys_groups_root_str
\c__keys_inherit_root_str
\c__keys_type_root_str
22450 \str_const:Nn \c__keys_code_root_str { key~code~>~ }
22451 \str_const:Nn \c__keys_check_root_str { key~check~>~ }
22452 \str_const:Nn \c__keys_default_root_str { key~default~>~ }
22453 \str_const:Nn \c__keys_groups_root_str { key~groups~>~ }
22454 \str_const:Nn \c__keys_inherit_root_str { key~inherit~>~ }
22455 \str_const:Nn \c__keys_type_root_str { key~type~>~ }
```

(End of definition for \c__keys_code_root_str and others.)

\c__keys_props_root_str

The prefix for storing properties.

```
22456 \str_const:Nn \c__keys_props_root_str { key~prop~>~ }
```

(End of definition for \c__keys_props_root_str.)

\l_keys_choice_int Publicly accessible data on which choice is being used when several are generated as a set.

\l_keys_choice_str

22457 `\int_new:N \l_keys_choice_int`
22458 `\str_new:N \l_keys_choice_str`

(End of definition for \l_keys_choice_int and \l_keys_choice_str. These variables are documented on page 252.)

\l_keys_choice_tl The `tl` version is deprecated but has to be handled manually.

22459 `\tl_new:N \l_keys_choice_tl`

(End of definition for \l_keys_choice_tl.)

\l__keys_exp_str The expansion postfix of a key name.

22460 `\str_new:N \l__keys_exp_str`

(End of definition for \l__keys_exp_str.)

\l__keys_groups_clist Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

22461 `\clist_new:N \l__keys_groups_clist`

(End of definition for \l__keys_groups_clist.)

\l__keys_inherit_bool For inheritance, particularly for recursion.

22462 `\bool_new:N \l__keys_inherit_bool`

(End of definition for \l__keys_inherit_bool.)

\l__keys_inherit_clist For normalization.

22463 `\clist_new:N \l__keys_inherit_clist`

(End of definition for \l__keys_inherit_clist.)

\l_keys_key_str The name of a key itself: needed when setting keys.

22464 `\str_new:N \l_keys_key_str`

(End of definition for \l_keys_key_str. This variable is documented on page 254.)

\l_keys_key_tl The `tl` version is deprecated but has to be handled manually.

22465 `\tl_new:N \l_keys_key_tl`

(End of definition for \l_keys_key_tl.)

\l__keys_module_str The module for an entire set of keys.

22466 `\str_new:N \l__keys_module_str`

(End of definition for \l__keys_module_str.)

\l__keys_no_value_bool A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

22467 `\bool_new:N \l__keys_no_value_bool`

(End of definition for \l__keys_no_value_bool.)

`\l__keys_only_known_bool` Used to track if only “known” keys are being set.
22468 `\bool_new:N \l__keys_only_known_bool`
(End of definition for \l__keys_only_known_bool.)

`\l_keys_path_str` The “path” of the current key is stored here: this is available to the programmer and so is public.
22469 `\str_new:N \l_keys_path_str`
(End of definition for \l_keys_path_str. This variable is documented on page 254.)

`\l_keys_path_tl` The older version is deprecated but has to be handled manually.
22470 `\tl_new:N \l_keys_path_tl`
(End of definition for \l_keys_path_tl.)

`\l__keys_inherit_str`
22471 `\str_new:N \l__keys_inherit_str`
(End of definition for \l__keys_inherit_str.)

`\l__keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.
22472 `\tl_new:N \l__keys_relative_tl`
22473 `\tl_set:Nn \l__keys_relative_tl { \q__keys_no_value }`
(End of definition for \l__keys_relative_tl.)

`\l__keys_property_str` The “property” begin set for a key at definition time is stored here.
22474 `\str_new:N \l__keys_property_str`
(End of definition for \l__keys_property_str.)

`\l__keys_selective_bool` Two booleans for using key groups: one to indicate that “selective” setting is active, a
`\l__keys_exclude_bool` second to specify which type (“opt-in” or “opt-out”).
22475 `\bool_new:N \l__keys_selective_bool`
22476 `\bool_new:N \l__keys_exclude_bool`
(End of definition for \l__keys_selective_bool and \l__keys_exclude_bool.)

`\l__keys_selective_clist` The list of key groups being filtered in or out during selective setting.
22477 `\clist_new:N \l__keys_selective_clist`
(End of definition for \l__keys_selective_clist.)

`\l__keys_tmp_clist` Scratch space used as a data dump.
22478 `\clist_new:N \l__keys_tmp_clist`
(End of definition for \l__keys_tmp_clist.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.
22479 `\clist_new:N \l__keys_unused_clist`
(End of definition for \l__keys_unused_clist.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

22480 `\tl_new:N \l_keys_value_tl`

(End of definition for `\l_keys_value_tl`. This variable is documented on page 254.)

`\l__keys_tmp_bool` Scratch space.

`\l__keys_tmpa_tl` 22481 `\bool_new:N \l__keys_tmp_bool`

`\l__keys_tmpb_tl` 22482 `\tl_new:N \l__keys_tmpa_tl`

22483 `\tl_new:N \l__keys_tmpb_tl`

(End of definition for `\l__keys_tmp_bool`, `\l__keys_tmpa_tl`, and `\l__keys_tmpb_tl`.)

`\l_keys_precompile_bool` For digesting keys.

`\l_keys_precompile_tl` 22484 `\bool_new:N \l_keys_precompile_bool`

22485 `\tl_new:N \l_keys_precompile_tl`

(End of definition for `\l_keys_precompile_bool` and `\l_keys_precompile_tl`.)

`\l_keys_usage_load_prop` Global data for document-level information.

`\l_keys_usage_preamble_prop` 22486 `\prop_new:N \l_keys_usage_load_prop`

22487 `\prop_new:N \l_keys_usage_preamble_prop`

(End of definition for `\l_keys_usage_load_prop` and `\l_keys_usage_preamble_prop`. These variables are documented on page 254.)

70.2.1 Internal auxiliaries

`\s__keys_nil` Internal scan marks.

`\s__keys_mark` 22488 `\scan_new:N \s__keys_nil`

`\s__keys_stop` 22489 `\scan_new:N \s__keys_mark`

22490 `\scan_new:N \s__keys_stop`

(End of definition for `\s__keys_nil`, `\s__keys_mark`, and `\s__keys_stop`.)

`\q__keys_no_value` Internal quarks.

22491 `\quark_new:N \q__keys_no_value`

(End of definition for `\q__keys_no_value`.)

`\__keys_quark_if_no_value_p:N` Branching quark conditional.

`\__keys_quark_if_no_value:NTF` 22492 `\__kernel_quark_new_conditional:Nn \__keys_quark_if_no_value:N { TF }`

(End of definition for `\__keys_quark_if_no_value:NTF`.)

`\__keys_precompile:n` An auxiliary to allow cleaner showing of code.

22493 `\cs_new_protected:Npn \__keys_precompile:n #1`

22494 `{`

22495 `\bool_if:NTF \l__keys_precompile_bool`

22496 `{ \tl_put_right:Nn \l__keys_precompile_tl }`

22497 `{ \use:n }`

22498 `{#1}`

22499 `}`

(End of definition for `\__keys_precompile:n`.)

`\__keys_cs_undefine:c` Local version of `\cs_undefine:c` to avoid sprinkling `\tex_undefined:D` everywhere.

```

22500 \cs_new_protected:Npn \__keys_cs_undefine:c #1
22501 {
22502   \if_cs_exist:w #1 \cs_end:
22503   \else:
22504     \use_i:nnnn
22505   \fi:
22506   \cs_set_eq:cN {#1} \tex_undefined:D
22507 }

```

(End of definition for `\__keys_cs_undefine:c`.)

70.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

`\keys_define:ne`
`\keys_define:nx`

```

22508 \cs_new_protected:Npn \keys_define:nn #1#2
22509 {
22510   \use:e
22511   {
22512     \exp_not:n
22513     {
22514       \str_set:Nl \l__keys_module_str { \__keys_trim_spaces:n {#1} }
22515       \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#2}
22516     }
22517     \__keys_reset_var:N \l__keys_module_str
22518     \__keys_reset_var:N \l__keys_inherit_str
22519     \__keys_reset_var:N \l__keys_choice_tl
22520     \__keys_reset_var:N \l__keys_choice_str
22521     \__keys_reset_var:N \l__keys_key_tl
22522     \__keys_reset_var:N \l__keys_key_str
22523     \__keys_reset_var:N \l__keys_path_tl
22524     \__keys_reset_var:N \l__keys_path_str
22525     \__keys_reset_var:N \l__keys_value_tl
22526     \int_set:Nn \l__keys_choice_int { \int_use:N \l__keys_choice_int }
22527   }
22528 }
22529 \cs_generate_variant:Nn \keys_define:nn { ne , nx }

```

(End of definition for `\keys_define:nn`. This function is documented on page 246.)

`\__keys_define:n` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

`\__keys_define:nn`
`\__keys_define_aux:nn`

```

22530 \cs_new_protected:Npn \__keys_define:n #1
22531 {
22532   \bool_set_true:N \l__keys_no_value_bool
22533   \__keys_define_aux:nn {#1} { }
22534 }
22535 \cs_new_protected:Npn \__keys_define:nn #1#2
22536 {
22537   \bool_set_false:N \l__keys_no_value_bool

```

```

22538     \__keys_define_aux:nn {#1} {#2}
22539   }
22540 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
22541 {
22542   \__keys_property_find:n {#1}
22543   \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
22544     { \__keys_define_code:n {#2} }
22545     {
22546       \str_if_empty:NF \l__keys_property_str
22547       {
22548         \msg_error:nnee { keys } { property-unknown }
22549         \l__keys_property_str \l_keys_path_str
22550       }
22551     }
22552 }

```

(End of definition for __keys_define:n, __keys_define:nn, and __keys_define_aux:nn.)

```

\__keys_property_find:n Searching for a property means finding the last . in the input, and storing the text before
\__keys_property_find_auxi:w and after it. Everything is first turned into strings, so there is no problem using \cs_
  \__keys_property_find_auxii:w set_nopar:Npe instead of \str_set:Np to set \l_keys_path_str. To gain further speed,
  \__keys_property_find_auxiii:w brace tricks are used and \__keys_property_find_auxiv:w is defined as expandable.
  \__keys_property_find_auxiv:w Since spaces will already be trimmed from the module we can omit it from the argument
\__keys_property_find_err:w to \__keys_trim_spaces:n.

22553 \cs_new_protected:Npn \__keys_property_find:n #1
22554 {
22555   \exp_after:wN \__keys_property_find_auxi:w \tl_to_str:n {#1}
22556   \s__keys_nil \__keys_property_find_auxii:w
22557   . \s__keys_nil \__keys_property_find_err:w
22558 }
22559 \cs_new:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
22560 {
22561   #3 #1 \s__keys_mark #2 \s__keys_nil #3
22562 }
22563 \cs_new_protected:Npn \__keys_property_find_auxii:w
22564 #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
22565 \__keys_property_find_err:w
22566 {
22567   \cs_set_nopar:Npe \l_keys_path_str
22568   {
22569     \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / }
22570     \exp_after:wN \__keys_trim_spaces:n \tex_expanded:D {
22571       #1
22572       \if_false: }}}} \fi:
22573       \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w
22574       . \s__keys_nil \__keys_property_find_auxiv:w
22575     }
22576 \cs_new:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark #2 . #3 \s__keys_nil #4
22577 {
22578   . #1 #4 #2 \s__keys_mark #3 \s__keys_nil #4
22579 }
22580 \cs_new:Npn \__keys_property_find_auxiv:w
22581 #1 \s__keys_nil \__keys_property_find_auxiii:w
22582 \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w

```

```

22583 {
22584   \if_false: {{\fi: }}
22585   \cs_set_nopar:Npe \l__keys_property_str { . #1 }
22586   \tl_set_eq:NN \l_keys_path_tl \l_keys_path_str
22587 }
22588 \cs_new_protected:Npn \__keys_property_find_err:w
22589 #1 \s_keys_nil #2 \__keys_property_find_err:w
22590 {
22591   \str_clear:N \l__keys_property_str
22592   \msg_error:nnn { keys } { no-property } {#1}
22593 }

```

(End of definition for __keys_property_find:n and others.)

__keys_define_code:n Two possible cases. If there is a value for the key, then just use the function. If not,
 __keys_define_code:nnn then a check to make sure there is no need for a value with the property. If there should
 __keys_define_code:w be one then complain, otherwise execute it. For a $\text{\LaTeX} 2_\epsilon$ property like .code which
 doesn't contain a :, treat it as having arity 1 and pass the (empty) value to it.

```

22594 \cs_new_protected:Npn \__keys_define_code:n #1
22595 {
22596   \bool_if:NTF \l__keys_no_value_bool
22597   {
22598     \__keys_define_code:nnn
22599     { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
22600     { \use:c { \c__keys_props_root_str \l__keys_property_str } }
22601     {
22602       \msg_error:nnee { keys } { property-requires-value }
22603       \l__keys_property_str \l_keys_path_str
22604     }
22605   }
22606   { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
22607 }
22608 \cs_new:Npe \__keys_define_code:nnn
22609 {
22610   \exp_not:N \exp_after:wN \exp_not:N \__keys_define_code:w
22611   \exp_not:N \l__keys_property_str
22612   \c_colon_str \c_colon_str
22613   \exp_not:N \s__keys_stop
22614 }
22615 \use:e
22616 {
22617   \cs_new:Npn \exp_not:N \__keys_define_code:w
22618   #1 \c_colon_str #2 \c_colon_str #3 \exp_not:N \s__keys_stop
22619 }
22620 {
22621   \tl_if_empty:nTF {#3}
22622   { \use_i:nnn }
22623   {
22624     \tl_if_empty:nTF {#2}
22625     { \use_ii:nnn }
22626     { \use_iii:nnn }
22627   }
22628 }

```

(End of definition for `\__keys_define_code:n`, `\__keys_define_code:nnn`, and `\__keys_define_code:w`.)

70.4 Turning properties into actions

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

\__keys_bool_set:Nn
\__keys_bool_set:cn
\__keys_bool_set_inverse:Nn
\__keys_bool_set_inverse:cn
\__keys_bool_set:Nnnn
22629 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
22630 { \__keys_bool_set:Nnnn #1 {#2} { true } { false } }
22631 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }
22632 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
22633 { \__keys_bool_set:Nnnn #1 {#2} { false } { true } }
22634 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
22635 \cs_new_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
22636 {
22637   \bool_if_exist:NF #1 { \bool_new:N #1 }
22638   \__keys_choice_make:
22639   \__keys_cmd_set:ne { \l_keys_path_str / true }
22640   { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
22641   \__keys_cmd_set:ne { \l_keys_path_str / false }
22642   { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
22643   \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
22644   {
22645     \msg_error:nne { keys } { boolean-values-only }
22646     \l_keys_path_str
22647   }
22648   \__keys_default_set:n { true }
22649 }
22650 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End of definition for `\__keys_bool_set:Nn`, `\__keys_bool_set_inverse:Nn`, and `\__keys_bool_set:Nnnn`.)

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoice and choices are essentially the same bar one function, the code is given together.

```

\__keys_choice_make:
\__keys_multichoice_make:
\__keys_choice_make:N
\__keys_choice_make_aux:N
22651 \cs_new_protected:Npn \__keys_choice_make:
22652 { \__keys_choice_make:N \__keys_choice_find:n }
22653 \cs_new_protected:Npn \__keys_multichoice_make:
22654 { \__keys_choice_make:N \__keys_multichoice_find:n }
22655 \cs_new_protected:Npn \__keys_choice_make:N #1
22656 {
22657   \cs_if_exist:cTF
22658   { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
22659   {
22660     \str_if_eq:vnTF
22661     { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
22662     { choice }
22663     {
22664       \msg_error:nnee { keys } { nested-choice-key }
22665       \l_keys_path_tl { \__keys_parent:o \l_keys_path_str }
22666     }
22667     { \__keys_choice_make_aux:N #1 }

```

```

22668     }
22669     { \_keys_choice_make_aux:N #1 }
22670   }
22671 \cs_new_protected:Npn \_keys_choice_make_aux:N #1
22672 {
22673   \cs_set_nopar:cpn { \c__keys_type_root_str \l_keys_path_str }
22674   { choice }
22675   \_keys_cmd_set_direct:nn \l_keys_path_str { #1 {##1} }
22676   \_keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
22677   {
22678     \msg_error:nnee { keys } { choice-unknown }
22679     \l_keys_path_str {##1}
22680   }
22681 }

```

(End of definition for _keys_choice_make: and others.)

_keys_choices_make:nn Auto-generating choices means setting up the root key as a choice, then defining each
_keys_multichoices_make:nn choice in turn.

```

\_keys_choices_make:Nnn
22682 \cs_new_protected:Npn \_keys_choices_make:nn
22683 { \_keys_choices_make:Nnn \_keys_choice_make: }
22684 \cs_new_protected:Npn \_keys_multichoices_make:nn
22685 { \_keys_choices_make:Nnn \_keys_multichoice_make: }
22686 \cs_new_protected:Npn \_keys_choices_make:Nnn #1#2#3
22687 {
22688   #1
22689   \int_zero:N \l_keys_choice_int
22690   \clist_map_inline:nn {#2}
22691   {
22692     \int_incr:N \l_keys_choice_int
22693     \_keys_cmd_set:ne
22694     { \l_keys_path_str / \_keys_trim_spaces:n {##1} }
22695     {
22696       \str_set:Nn \exp_not:N \l_keys_choice_str {##1}
22697       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
22698       \int_set:Nn \exp_not:N \l_keys_choice_int
22699       { \int_use:N \l_keys_choice_int }
22700       \exp_not:n {#3}
22701     }
22702   }
22703 }

```

(End of definition for _keys_choices_make:nn, _keys_multichoices_make:nn, and _keys_choices_make:Nnn.)

_keys_cmd_set:nn Setting the code for a key first logs if appropriate that we are defining a new key, then
_keys_cmd_set:Vn saves the code.

```

\_keys_cmd_set:ne
\_keys_cmd_set:Vo
\_keys_cmd_set_direct:nn
22704 \cs_new_protected:Npn \_keys_cmd_set:nn #1#2
22705 { \_keys_cmd_set_direct:nn {#1} { \_keys_precompile:n {#2} } }
22706 \cs_generate_variant:Nn \_keys_cmd_set:nn { ne , Vn , Vo }
22707 \cs_new_protected:Npn \_keys_cmd_set_direct:nn #1#2
22708 { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }

```

(End of definition for _keys_cmd_set:nn and _keys_cmd_set_direct:nn.)

`\__keys_cs_set:NNpn` Creating control sequences is a bit more tricky than other cases as we need to pick up the `p` argument. To make the internals look clearer, the trailing `n` argument here is just for appearance.

```

22709 \cs_new_protected:Npn \__keys_cs_set:NNpn #1#2#3#
22710 {
22711   \cs_set_protected:cpe { \c__keys_code_root_str \l_keys_path_str } ##1
22712   {
22713     \__keys_precompile:n
22714     { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
22715   }
22716   \use_none:n
22717 }
22718 \cs_generate_variant:Nn \__keys_cs_set:NNpn { Nc }

```

(End of definition for __keys_cs_set:NNpn.)

`\__keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set_nopar:cpe` as this avoids any worries about whether a token list exists.

```

22719 \cs_new_protected:Npn \__keys_default_set:n #1
22720 {
22721   \tl_if_empty:nTF {#1}
22722   {
22723     \__keys_cs_undefine:c
22724     { \c__keys_default_root_str \l_keys_path_str }
22725   }
22726   {
22727     \cs_set_nopar:cpe
22728     { \c__keys_default_root_str \l_keys_path_str }
22729     { \exp_not:n {#1} }
22730     \__keys_value_requirement:nn { required } { false }
22731   }
22732 }

```

(End of definition for __keys_default_set:n.)

`\__keys_groups_set:n` Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```

22733 \cs_new_protected:Npn \__keys_groups_set:n #1
22734 {
22735   \clist_set:Ne \l__keys_groups_clist { \tl_to_str:n {#1} }
22736   \clist_if_empty:NTF \l__keys_groups_clist
22737   {
22738     \__keys_cs_undefine:c
22739     { \c__keys_groups_root_str \l_keys_path_str }
22740   }
22741   {
22742     \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
22743     \l__keys_groups_clist
22744   }
22745 }

```

(End of definition for __keys_groups_set:n.)

`\\_keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

22746 \cs_new_protected:Npn \\_keys_inherit:n #1
22747 {
22748   \\_keys_undefine:
22749   \clist_set:Nn \\_keys_inherit_clist {#1}
22750   \cs_set_eq:cN { \\_keys_inherit_root_str \\l_keys_path_str }
22751   \\_keys_inherit_clist
22752 }

```

(End of definition for `\\_keys_inherit:n`.)

`\\_keys_initialize:n` A set up for initialization: just run the code if it exists. We need to set the key string here, using the deprecated `tl` var as a piece of scratch space. We need to test *first* for the key in the current tree then any inheritance.

```

22753 \cs_new_protected:Npn \\_keys_initialize:n #1
22754 {
22755   \exp_after:wN \\_keys_find_key_module:wNN
22756   \\l_keys_path_str \\s_keys_stop
22757   \\l_keys_key_tl \\l_keys_key_str
22758   \tl_set_eq:NN \\l_keys_key_tl \\l_keys_key_str
22759   \tl_set:Nn \\l_keys_value_tl {#1}
22760   \cs_if_exist:cTF { \\_keys_code_root_str \\l_keys_path_str }
22761   {
22762     \str_clear:N \\_keys_inherit_str
22763     \\_keys_execute:nn \\l_keys_path_str {#1}
22764   }
22765   {
22766     \cs_if_exist:cT
22767     { \\_keys_inherit_root_str \\_keys_parent:o \\l_keys_path_str }
22768     { \\_keys_execute_inherit: }
22769   }
22770 }

```

(End of definition for `\\_keys_initialize:n`.)

`\\_keys_legacy_if_set:nn` Much the same as `expl3` booleans, except we assume that the switch exists.

```

\\_keys_legacy_if_inverse:nn
\\_keys_legacy_if_inverse:nnnn
22771 \cs_new_protected:Npn \\_keys_legacy_if_set:nn #1#2
22772 { \\_keys_legacy_if_set:nnnn {#1} {#2} { true } { false } }
22773 \cs_new_protected:Npn \\_keys_legacy_if_set_inverse:nn #1#2
22774 { \\_keys_legacy_if_set:nnnn {#1} {#2} { false } { true } }
22775 \cs_new_protected:Npn \\_keys_legacy_if_set:nnnn #1#2#3#4
22776 {
22777   \\_keys_choice_make:
22778   \\_keys_cmd_set:ne { \\l_keys_path_str / true }
22779   { \exp_not:c { legacy_if_#2 set_ #3 :n } { \exp_not:n {#1} } }
22780   \\_keys_cmd_set:ne { \\l_keys_path_str / false }
22781   { \exp_not:c { legacy_if_#2 set_ #4 :n } { \exp_not:n {#1} } }
22782   \\_keys_cmd_set:nn { \\l_keys_path_str / unknown }
22783   {
22784     \msg_error:nne { keys } { boolean-values-only }
22785     \\l_keys_path_str
22786   }
22787   \\_keys_default_set:n { true }
22788   \cs_if_exist:cF { if#1 }

```

```

22789     {
22790         \cs:w newif \exp_after:wN \cs_end:
22791         \cs:w if#1 \cs_end:
22792     }
22793 }

```

(End of definition for __keys_legacy_if_set:nn, __keys_legacy_if_inverse:nn, and __keys_legacy_if_inverse:nnnn.)

__keys_meta_make:n To create a meta-key, simply set up to pass data through. The internal function is used here as a meta key should respect the prevailing filtering, etc.

```

\__keys_meta_make:nn
22794 \cs_new_protected:Npn \__keys_meta_make:n
22795 { \exp_args:NV \__keys_meta_make:nn \l__keys_module_str }
22796 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
22797 {
22798     \exp_args:NV \__keys_cmd_set_direct:nn
22799     \l__keys_path_str { \__keys_set:nn {#1} {#2} }
22800 }

```

(End of definition for __keys_meta_make:n and __keys_meta_make:nn.)

__keys_prop_put:Nn Much the same as other variables, but needs a dedicated auxiliary.

```

\__keys_prop_put:cn
22801 \cs_new_protected:Npn \__keys_prop_put:Nn #1#2
22802 {
22803     \prop_if_exist:NF #1 { \prop_new:N #1 }
22804     \exp_after:wN \__keys_find_key_module:wNN \l__keys_path_str \s__keys_stop
22805     \l__keys_tmpa_tl \l__keys_tmpb_tl
22806     \__keys_cmd_set:ne \l__keys_path_str
22807     {
22808         \exp_not:c { prop_ #2 put:Nnn }
22809         \exp_not:N #1
22810         { \l__keys_tmpb_tl }
22811         \exp_not:n { {##1} }
22812     }
22813 }
22814 \cs_generate_variant:Nn \__keys_prop_put:Nn { c }

```

(End of definition for __keys_prop_put:Nn.)

__keys_undefine: Undefining a key has to be done without \cs_undefine:c as that function acts globally.

```

22815 \cs_new_protected:Npn \__keys_undefine:
22816 {
22817     \clist_map_inline:nn
22818     { code , default , groups , inherit , type , check }
22819     {
22820         \__keys_cs_undefine:c
22821         { \tl_use:c { c__keys_ ##1 _root_str } \l__keys_path_str }
22822     }
22823 }

```

(End of definition for __keys_undefine:.)

Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional cname per key that needs it. The cleanup here has to know the structure of the following code.

```

22824 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
22825 {
22826   \str_case:nnF {#2}
22827   {
22828     { true }
22829     {
22830       \cs_set_eq:cc
22831       { \c__keys_check_root_str \l_keys_path_str }
22832       { __keys_check_ #1 : }
22833     }
22834     { false }
22835     {
22836       \cs_if_eq:ccT
22837       { \c__keys_check_root_str \l_keys_path_str }
22838       { __keys_check_ #1 : }
22839       {
22840         \__keys_cs_undefine:c
22841         { \c__keys_check_root_str \l_keys_path_str }
22842       }
22843     }
22844   }
22845   {
22846     \msg_error:nne { keys }
22847     { boolean-values-only }
22848     { .value_ #1 :n }
22849   }
22850 }
22851 \cs_new_protected:Npn \__keys_check_forbidden:
22852 {
22853   \bool_if:NF \l__keys_no_value_bool
22854   {
22855     \msg_error:nnee { keys } { value-forbidden }
22856     \l_keys_path_str \l_keys_value_tl
22857     \use_none:nnn
22858   }
22859 }
22860 \cs_new_protected:Npn \__keys_check_required:
22861 {
22862   \bool_if:NT \l__keys_no_value_bool
22863   {
22864     \msg_error:nne { keys } { value-required }
22865     \l_keys_path_str
22866     \use_none:nnn
22867   }
22868 }

```

(End of definition for __keys_value_requirement:nn, __keys_check_forbidden:, and __keys_check_required:.)

```

\__keys_usage:n Save the relevant data.
\__keys_usage:NN
\__keys_usage:w
\__keys_usage_aux:w

```

```

22869 \cs_new_protected:Npn \__keys_usage:n #1
22870 {
22871   \str_case:nnF {#1}
22872   {
22873     { general }
22874     {
22875       \__keys_usage:NN \l_keys_usage_load_prop
22876       \c_false_bool
22877       \__keys_usage:NN \l_keys_usage_preamble_prop
22878       \c_false_bool
22879     }
22880   { load }
22881   {
22882     \__keys_usage:NN \l_keys_usage_load_prop
22883     \c_true_bool
22884     \__keys_usage:NN \l_keys_usage_preamble_prop
22885     \c_false_bool
22886   }
22887   { preamble }
22888   {
22889     \__keys_usage:NN \l_keys_usage_load_prop
22890     \c_false_bool
22891     \__keys_usage:NN \l_keys_usage_preamble_prop
22892     \c_true_bool
22893   }
22894 }
22895 {
22896   \msg_error:nnnn { keys }
22897   { choice-unknown }
22898   { .usage:n }
22899   {#1}
22900 }
22901 }
22902 \cs_new_protected:Npn \__keys_usage:NN #1#2
22903 {
22904   \prop_get:NVNF #1 \l__keys_module_str \l__keys_tmpa_tl
22905   { \tl_clear:N \l__keys_tmpa_tl }
22906   \tl_set:Ne \l__keys_tmpb_tl
22907   {
22908     \exp_after:wN \exp_after:wN \exp_after:wN \__keys_usage:w \exp_after:wN
22909     \l_keys_path_str \exp_after:wN / \exp_after:wN \s__keys_stop
22910     \exp_after:wN { \l_keys_path_str }
22911   }
22912   \bool_if:NTF #2
22913   { \clist_put_right:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
22914   { \clist_remove_all:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
22915   \prop_put:NVV #1 \l__keys_module_str
22916   \l__keys_tmpa_tl
22917 }
22918 \cs_new:Npn \__keys_usage:w #1 / #2 \s__keys_stop #3
22919 {
22920   \tl_if_blank:nTF {#2}
22921   {#1}
22922   { \__keys_usage_aux:w #3 \s__keys_stop }

```

```

22923     }
22924 \cs_new:Npn \__keys_usage_aux:w #1 / #2 \s__keys_stop {#2}

```

(End of definition for __keys_usage:n and others.)

__keys_variable_set:NnnN Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```

\__keys_variable_set:cnnN
\__keys_variable_set_required:NnnN
\__keys_variable_set_required:cnnN
22925 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
22926 {
22927     \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
22928     \__keys_cmd_set:ne \l_keys_path_str
22929     {
22930         \exp_not:c { #2 _ #3 set:N #4 }
22931         \exp_not:N #1
22932         \exp_not:n { {#1} }
22933     }
22934 }
22935 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
22936 \cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
22937 {
22938     \__keys_variable_set:NnnN #1 {#2} {#3} #4
22939     \__keys_value_requirement:nn { required } { true }
22940 }
22941 \cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }

```

(End of definition for __keys_variable_set:NnnN and __keys_variable_set_required:NnnN.)

70.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

```

.bool_set:N One function for this.
.bool_set:c 22942 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
22943 { \__keys_bool_set:Nn #1 { } }
.bool_gset:N 22944 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
22945 { \__keys_bool_set:cn {#1} { } }
.bool_gset:c 22946 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
22947 { \__keys_bool_set:Nn #1 { g } }
22948 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
22949 { \__keys_bool_set:cn {#1} { g } }

```

(End of definition for .bool_set:N and .bool_gset:N. These functions are documented on page 247.)

```

.bool_set_inverse:N One function for this.
.bool_set_inverse:c 22950 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:N } #1
22951 { \__keys_bool_set_inverse:Nn #1 { } }
.bool_gset_inverse:N 22952 \cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:c } #1
22953 { \__keys_bool_set_inverse:cn {#1} { } }

```

```

22954 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:N } #1
22955   { \__keys_bool_set_inverse:Nn #1 { g } }
22956 \cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:c } #1
22957   { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End of definition for `.bool_set_inverse:N` and `.bool_gset_inverse:N`. These functions are documented on page 247.)

.choice: Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

22958 \cs_new_protected:cpn { \c__keys_props_root_str .choice: }
22959   { \__keys_choice_make: }

```

(End of definition for `.choice:`. This function is documented on page 247.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

```

.choices:Vn
.choices:en
.choices:on
.choices:xn
22960 \cs_new_protected:cpn { \c__keys_props_root_str .choices:nn } #1
22961   { \__keys_choices_make:nn #1 }
22962 \cs_new_protected:cpn { \c__keys_props_root_str .choices:Vn } #1
22963   { \exp_args:NV \__keys_choices_make:nn #1 }
22964 \cs_new_protected:cpn { \c__keys_props_root_str .choices:en } #1
22965   { \exp_args:Ne \__keys_choices_make:nn #1 }
22966 \cs_new_protected:cpn { \c__keys_props_root_str .choices:on } #1
22967   { \exp_args:No \__keys_choices_make:nn #1 }
22968 \cs_new_protected:cpn { \c__keys_props_root_str .choices:xn } #1
22969   { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End of definition for `.choices:nn`. This function is documented on page 247.)

.code:n Creating code is simply a case of passing through to the underlying `set` function.

```

22970 \cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
22971   { \__keys_cmd_set:nn \l_keys_path_str {#1} }

```

(End of definition for `.code:n`. This function is documented on page 248.)

.clist_set:N

```

.clist_set:c
.clist_gset:N
.clist_gset:c
22972 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
22973   { \__keys_variable_set:NnnN #1 { clist } { } n }
22974 \cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
22975   { \__keys_variable_set:cnnN {#1} { clist } { } n }
22976 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
22977   { \__keys_variable_set:NnnN #1 { clist } { g } n }
22978 \cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
22979   { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End of definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 247.)

.cs_set:Np

```

.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp
22980 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
22981   { \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
22982 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
22983   { \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
22984 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
22985   { \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
22986 \cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1

```

```

22987 { \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
22988 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
22989 { \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
22990 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
22991 { \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
22992 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
22993 { \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
22994 \cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
22995 { \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }

```

(End of definition for `.cs_set:Np` and others. These functions are documented on page 248.)

.default:n Expansion is left to the internal functions.

```

.default:V 22996 \cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
.default:e 22997 { \__keys_default_set:n {#1} }
.default:o 22998 \cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
.default:x 22999 { \exp_args:NV \__keys_default_set:n #1 }
23000 \cs_new_protected:cpn { \c__keys_props_root_str .default:e } #1
23001 { \exp_args:Ne \__keys_default_set:n {#1} }
23002 \cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
23003 { \exp_args:No \__keys_default_set:n {#1} }
23004 \cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
23005 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End of definition for `.default:n`. This function is documented on page 248.)

.dim_set:N Setting a variable is very easy: just pass the data along.

```

.dim_set:c 23006 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
.dim_gset:N 23007 { \__keys_variable_set_required:NnnN #1 { dim } { } n }
.dim_gset:c 23008 \cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
23009 { \__keys_variable_set_required:cnnN {#1} { dim } { } n }
23010 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
23011 { \__keys_variable_set_required:NnnN #1 { dim } { g } n }
23012 \cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
23013 { \__keys_variable_set_required:cnnN {#1} { dim } { g } n }

```

(End of definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 248.)

.fp_set:N Setting a variable is very easy: just pass the data along.

```

.fp_set:c 23014 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1
.fp_gset:N 23015 { \__keys_variable_set_required:NnnN #1 { fp } { } n }
.fp_gset:c 23016 \cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1
23017 { \__keys_variable_set_required:cnnN {#1} { fp } { } n }
23018 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1
23019 { \__keys_variable_set_required:NnnN #1 { fp } { g } n }
23020 \cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1
23021 { \__keys_variable_set_required:cnnN {#1} { fp } { g } n }

```

(End of definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 248.)

.groups:n A single property to create groups of keys.

```

23022 \cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1
23023 { \__keys_groups_set:n {#1} }

```

(End of definition for `.groups:n`. This function is documented on page 249.)

.inherit:n Nothing complex: only one variant at the moment!

```
23024 \cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1
23025 { \__keys_inherit:n {#1} }
```

(End of definition for .inherit:n. This function is documented on page 249.)

.initial:n The standard hand-off approach.

```
.initial:V 23026 \cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1
.initial:e 23027 { \__keys_initialize:n {#1} }
.initial:o 23028 \cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1
.initial:x 23029 { \exp_args:NV \__keys_initialize:n #1 }
23030 \cs_new_protected:cpn { \c__keys_props_root_str .initial:e } #1
23031 { \exp_args:Ne \__keys_initialize:n {#1} }
23032 \cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1
23033 { \exp_args:No \__keys_initialize:n {#1} }
23034 \cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1
23035 { \exp_args:Nx \__keys_initialize:n {#1} }
```

(End of definition for .initial:n. This function is documented on page 249.)

.int_set:N Setting a variable is very easy: just pass the data along.

```
.int_set:c 23036 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1
.int_gset:N 23037 { \__keys_variable_set_required:NnnN #1 { int } { } n }
.int_gset:c 23038 \cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1
23039 { \__keys_variable_set_required:cnnN {#1} { int } { } n }
23040 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1
23041 { \__keys_variable_set_required:NnnN #1 { int } { g } n }
23042 \cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1
23043 { \__keys_variable_set_required:cnnN {#1} { int } { g } n }
```

(End of definition for .int_set:N and .int_gset:N. These functions are documented on page 249.)

```
.legacy_if_set:n
.legacy_if_gset:n 23044 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set:n } #1
.legacy_if_set_inverse:n 23045 { \__keys_legacy_if_set:nn {#1} { } }
.legacy_if_gset_inverse:n 23046 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset:n } #1
23047 { \__keys_legacy_if_set:nn {#1} { g } }
23048 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set_inverse:n } #1
23049 { \__keys_legacy_if_set_inverse:nn {#1} { } }
23050 \cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset_inverse:n } #1
23051 { \__keys_legacy_if_set_inverse:nn {#1} { g } }
```

(End of definition for .legacy_if_set:n and others. These functions are documented on page 249.)

.meta:n Making a meta is handled internally.

```
23052 \cs_new_protected:cpn { \c__keys_props_root_str .meta:n } #1
23053 { \__keys_meta_make:n {#1} }
```

(End of definition for .meta:n. This function is documented on page 249.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
23054 \cs_new_protected:cpn { \c__keys_props_root_str .meta:nn } #1
23055 { \__keys_meta_make:nn #1 }
```

(End of definition for .meta:nn. This function is documented on page 250.)

.multichoice: The same idea as **.choice:** and **.choices:nn**, but where more than one choice is allowed.

```
.multichoices:nn 23056 \cs_new_protected:cpn { \c__keys_props_root_str .multichoice: }
                23057 { \__keys_multichoice_make: }
.multichoices:Vn 23058 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:nn } #1
                23059 { \__keys_multichoices_make:nn #1 }
.multichoices:en 23060 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:Vn } #1
                23061 { \exp_args:NV \__keys_multichoices_make:nn #1 }
.multichoices:on 23062 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:en } #1
                23063 { \exp_args:Ne \__keys_multichoices_make:nn #1 }
                23064 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
                23065 { \exp_args:No \__keys_multichoices_make:nn #1 }
                23066 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
                23067 { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(End of definition for **.multichoice:** and **.multichoices:nn**. These functions are documented on page 250.)

.muskip_set:N Setting a variable is very easy: just pass the data along.

```
.muskip_set:c 23068 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
                { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
.muskip_gset:N 23069 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
                { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
.muskip_gset:c 23070 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
                { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
                23071 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
                { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }
```

(End of definition for **.muskip_set:N** and **.muskip_gset:N**. These functions are documented on page 250.)

.prop_put:N Setting a variable is very easy: just pass the data along.

```
.prop_put:c 23076 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
                { \__keys_prop_put:Nn #1 { } }
.prop_gput:N 23077 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
                { \__keys_prop_put:cn {#1} { } }
.prop_gput:c 23078 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
                { \__keys_prop_put:Nn #1 { g } }
                23079 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
                { \__keys_prop_put:cn {#1} { g } }
```

(End of definition for **.prop_put:N** and **.prop_gput:N**. These functions are documented on page 250.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```
.skip_set:c 23084 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
                { \__keys_variable_set_required:NnnN #1 { skip } { } n }
.skip_gset:N 23085 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
                { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
                23086 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
                { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
                23087 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
                { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }
```

(End of definition for **.skip_set:N** and **.skip_gset:N**. These functions are documented on page 250.)

```

.str_set:N Setting a variable is very easy: just pass the data along.
.str_set:c 23092 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:N } #1
.str_gset:N 23093 { \__keys_variable_set:NnnN #1 { str } { } n }
.str_gset:c 23094 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:c } #1
.str_set_e:N 23095 { \__keys_variable_set:cnnN {#1} { str } { } n }
.str_set_e:c 23096 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:N } #1
.str_gset_e:N 23097 { \__keys_variable_set:NnnN #1 { str } { } e }
.str_gset_e:c 23098 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:c } #1
23099 { \__keys_variable_set:cnnN {#1} { str } { } e }
23100 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:N } #1
23101 { \__keys_variable_set:NnnN #1 { str } { g } n }
23102 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:c } #1
23103 { \__keys_variable_set:cnnN {#1} { str } { g } n }
23104 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:N } #1
23105 { \__keys_variable_set:NnnN #1 { str } { g } e }
23106 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:c } #1
23107 { \__keys_variable_set:cnnN {#1} { str } { g } e }

```

(End of definition for .str_set:N and others. These functions are documented on page 250.)

```

.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c 23108 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
.tl_gset:N 23109 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:c 23110 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
.tl_set_e:N 23111 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_e:c 23112 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:N } #1
.tl_gset_e:N 23113 { \__keys_variable_set:NnnN #1 { tl } { } e }
.tl_gset_e:c 23114 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:c } #1
23115 { \__keys_variable_set:cnnN {#1} { tl } { } e }
23116 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1
23117 { \__keys_variable_set:NnnN #1 { tl } { g } n }
23118 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
23119 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
23120 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:N } #1
23121 { \__keys_variable_set:NnnN #1 { tl } { g } e }
23122 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:c } #1
23123 { \__keys_variable_set:cnnN {#1} { tl } { g } e }

```

(End of definition for .tl_set:N and others. These functions are documented on page 250.)

```

.undefine: Another simple wrapper.
23124 \cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
23125 { \__keys_undefine: }

```

(End of definition for .undefine:. This function is documented on page 251.)

```

.usage:n
23126 \cs_new_protected:cpn { \c__keys_props_root_str .usage:n } #1
23127 { \__keys_usage:n {#1} }

```

(End of definition for .usage:n. This function is documented on page 254.)

`.value_forbidden:n`
`.value_required:n`

These are very similar, so both call the same function.

```
23128 \cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
23129 { \__keys_value_requirement:nn { forbidden } {#1} }
23130 \cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
23131 { \__keys_value_requirement:nn { required } {#1} }
```

(End of definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page 251.)

70.6 Setting keys

`\__keys_set:nnnnNn`
`\__keys_reset_bool:N`
`\__keys_reset_var:N`
`\__keys_set:nn`
`\__keys_set:nnn`

The aim here is to allow nesting of key setting without needing lots of tracking. That is done by expanding the appropriate tokens “around” the core keyval parsing. As there are several different sub-paths, this needs a few steps and some generic auxiliaries. The arguments here are

1. The root for keys
2. The key groups
3. The keys themselves
4. The relative root for return of unset keys
5. The `clist` var for returning unset keys
6. The code to set up the correct selection approach

```
23132 \cs_new_protected:Npn \__keys_set:nnnnNn #1#2#3#4#5#6
23133 {
23134   \use:e
23135   {
23136     \exp_not:n
23137     {
23138       \clist_clear:N \l__keys_unused_clist
23139       \clist_set:Nc \l__keys_selective_clist { \tl_to_str:n {#2} }
23140       \tl_set:Nn \l__keys_relative_tl {#4}
23141       #6
23142       \__keys_set:nn {#1} {#3}
23143       \clist_set_eq:NN #5 \l__keys_unused_clist
23144     }
23145     \__keys_reset_bool:N \l__keys_only_known_bool
23146     \__keys_reset_bool:N \l__keys_exclude_bool
23147     \__keys_reset_bool:N \l__keys_selective_bool
23148     \__keys_reset_var:N \l__keys_unused_clist
23149     \__keys_reset_var:N \l__keys_selective_clist
23150     \__keys_reset_var:N \l__keys_relative_tl
23151     \__keys_reset_var:N \l__keys_inherit_str
23152     \__keys_reset_var:N \l__keys_choice_tl
23153     \__keys_reset_var:N \l__keys_choice_str
23154     \__keys_reset_var:N \l__keys_key_tl
23155     \__keys_reset_var:N \l__keys_key_str
23156     \__keys_reset_var:N \l__keys_path_tl
23157     \__keys_reset_var:N \l__keys_path_str
23158     \__keys_reset_var:N \l__keys_value_tl
```

```

23159         \int_set:Nn \l_keys_choice_int { \int_use:N \l_keys_choice_int }
23160     }
23161 }
23162 \cs_new:Npn \__keys_reset_bool:N #1
23163 {
23164     \exp_not:c
23165     { bool_set_ \bool_if:NTF #1 { true } { false } :N }
23166     \exp_not:N #1
23167 }
23168 \cs_new:Npn \__keys_reset_var:N #1
23169 {
23170     \exp_not:n
23171     { \__kernel_tl_set:Nx #1 }
23172     { \exp_not:N \exp_not:n { \exp_not:o { #1 } } }
23173 }
23174 \cs_new_protected:Npn \__keys_set:nn #1#2
23175 { \exp_args:No \__keys_set:nnn \l__keys_module_str {#1} {#2} }
23176 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
23177 {
23178     \str_set:Ne \l__keys_module_str { \__keys_trim_spaces:n {#2} }
23179     \keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
23180     \str_set:Nn \l__keys_module_str {#1}
23181 }

```

(End of definition for __keys_set:nnnnNn and others.)

\keys_set:nn A simple wrapper allowing for nesting.

```

\keys_set:nV 23182 \cs_new_protected:Npn \keys_set:nn #1#2
\keys_set:nv 23183 {
\keys_set:ne 23184     \__keys_set:nnnnNn
\keys_set:no 23185     {#1} { } {#2} { \q_keys_no_value } \l__keys_tmp_clist
\keys_set:nx 23186     {
23187         \bool_set_false:N \l__keys_only_known_bool
23188         \bool_set_false:N \l__keys_exclude_bool
23189         \bool_set_false:N \l__keys_selective_bool
23190     }
23191 }
23192 \cs_generate_variant:Nn \keys_set:nn { nV , nv , ne , no , nx }

```

(End of definition for \keys_set:nn. This function is documented on page 254.)

\keys_set_known:nnnN Simply set the right variables.

```

\keys_set_known:nVnN 23193 \cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
\keys_set_known:nvnN 23194 {
\keys_set_known:nenN 23195     \__keys_set:nnnnNn
\keys_set_known:nonN 23196     {#1} { } {#2} {#3} #4
\keys_set_known:nnN 23197     {
23198         \bool_set_true:N \l__keys_only_known_bool
23199         \bool_set_false:N \l__keys_exclude_bool
23200         \bool_set_false:N \l__keys_selective_bool
23201     }
23202 }
\keys_set_known:nn 23203 \cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , ne , no }
\keys_set_known:nV 23204 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
\keys_set_known:nv 23205 { \keys_set_known:nnnN {#1} {#2} { \q_keys_no_value } #3 }
\keys_set_known:ne
\keys_set_known:no

```

```

23206 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , ne , no }
23207 \cs_new_protected:Npn \keys_set_known:nn #1#2
23208 { \keys_set_known:nnnN {#1} {#2} { \q__keys_no_value } \l__keys_tmp_clist }
23209 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , ne , no }

```

(End of definition for \keys_set_known:nnnN, \keys_set_known:nnN, and \keys_set_known:nn. These functions are documented on page 256.)

The same for (exclusion) groups.

```

\keys_set_exclude_groups:nnnN
\keys_set_exclude_groups:nnVN
\keys_set_exclude_groups:nnvN
\keys_set_exclude_groups:nnoN
\keys_set_exclude_groups:nnnnN
\keys_set_exclude_groups:nnVnN
\keys_set_exclude_groups:nnvnN
\keys_set_exclude_groups:nnonN
\keys_set_exclude_groups:nnn
\keys_set_exclude_groups:nnV
\keys_set_exclude_groups:nnv
\keys_set_exclude_groups:nno
\keys_set_groups:nnnN
\keys_set_groups:nnVN
\keys_set_groups:nnvN
\keys_set_groups:nnoN
\keys_set_groups:nnnnN
\keys_set_groups:nnVnN
\keys_set_groups:nnvnN
\keys_set_groups:nnonN
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
23210 \cs_new_protected:Npn \keys_set_exclude_groups:nnnnN #1#2#3#4#5
23211 {
23212   \__keys_set:nnnnNn
23213   {#1} {#2} {#3} {#4} #5
23214   {
23215     \bool_set_false:N \l__keys_only_known_bool
23216     \bool_set_true:N \l__keys_exclude_bool
23217     \bool_set_true:N \l__keys_selective_bool
23218   }
23219 }
23220 \cs_generate_variant:Nn \keys_set_exclude_groups:nnnnN { nnV , nnv , nno }
23221 \cs_new_protected:Npn \keys_set_exclude_groups:nnnN #1#2#3#4
23222 { \keys_set_exclude_groups:nnnnN {#1} {#2} {#3} { \q__keys_no_value } #4 }
23223 \cs_generate_variant:Nn \keys_set_exclude_groups:nnnN { nnV , nnv , nno }
23224 \cs_new_protected:Npn \keys_set_exclude_groups:nnn #1#2#3
23225 {
23226   \keys_set_exclude_groups:nnnnN {#1} {#2} {#3}
23227   { \q__keys_no_value } \l__keys_tmp_clist
23228 }
23229 \cs_generate_variant:Nn \keys_set_exclude_groups:nnn { nnV , nnv , nno }
23230 \cs_new_protected:Npn \keys_set_groups:nnnnN #1#2#3#4#5
23231 {
23232   \__keys_set:nnnnNn
23233   {#1} {#2} {#3} {#4} #5
23234   {
23235     \bool_set_false:N \l__keys_only_known_bool
23236     \bool_set_false:N \l__keys_exclude_bool
23237     \bool_set_true:N \l__keys_selective_bool
23238   }
23239 }
23240 \cs_generate_variant:Nn \keys_set_groups:nnnnN { nnV , nnv , nno }
23241 \cs_new_protected:Npn \keys_set_groups:nnnN #1#2#3#4
23242 { \keys_set_groups:nnnnN {#1} {#2} {#3} { \q__keys_no_value } #4 }
23243 \cs_generate_variant:Nn \keys_set_groups:nnnN { nnV , nnv , nno }
23244 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
23245 {
23246   \keys_set_groups:nnnnN {#1} {#2} {#3}
23247   { \q__keys_no_value } \l__keys_tmp_clist
23248 }
23249 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }

```

(End of definition for \keys_set_exclude_groups:nnnN and others. These functions are documented on page 257.)

\keys_precompile:nnN A simple wrapper.

```

23250 \cs_new_protected:Npn \keys_precompile:nnN #1#2#3

```

```

23251 {
23252     \bool_set_true:N \l__keys_precompile_bool
23253     \tl_clear:N \l__keys_precompile_tl
23254     \keys_set:nn {#1} {#2}
23255     \bool_set_false:N \l__keys_precompile_bool
23256     \tl_set_eq:NN #3 \l__keys_precompile_tl
23257 }

```

(End of definition for \keys_precompile:nnN. This function is documented on page 257.)

```

\__keys_set_keyval:n A shared system once again. First, set the current path and add a default if needed.
\__keys_set_keyval:nn There are then checks to see if a value is required or forbidden. If everything passes,
\__keys_set_keyval:nnn move on to execute the code.
\__keys_set_keyval:onn
\__keys_find_key_module:wNN
\__keys_find_key_module_auxi:Nw
\__keys_find_key_module_auxii:Nw
\__keys_find_key_module_auxiii:Nn
\__keys_find_key_module_auxiv:Nw
\__keys_find_key_module_auxv:Nw
\__keys_set_selective:
23258 \cs_new_protected:Npn \__keys_set_keyval:n #1
23259 {
23260     \bool_set_true:N \l__keys_no_value_bool
23261     \__keys_set_keyval:onn \l__keys_module_str {#1} { }
23262 }
23263 \cs_new_protected:Npn \__keys_set_keyval:nn #1#2
23264 {
23265     \bool_set_false:N \l__keys_no_value_bool
23266     \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
23267 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

23268 \cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
23269 {
23270     \__kernel_tl_set:Nx \l_keys_path_str
23271     {
23272         \tl_if_blank:nF {#1}
23273         { #1 / }
23274         \__keys_trim_spaces:n {#2}
23275     }
23276     \str_clear:N \l__keys_module_str
23277     \str_clear:N \l__keys_inherit_str
23278     \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
23279     \l__keys_module_str \l_keys_key_str
23280     \str_set:Ne \l_keys_path_str
23281     {
23282         \l__keys_module_str
23283         \str_if_empty:NF \l__keys_module_str { / }
23284         \l_keys_key_str
23285     }
23286     \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
23287     \__keys_value_or_default:n {#3}
23288     \bool_if:NTF \l__keys_selective_bool
23289     \__keys_set_selective:
23290     \__keys_execute:
23291     \str_set:Nn \l__keys_module_str {#1}
23292 }
23293 \cs_generate_variant:Nn \__keys_set_keyval:nnn { o }

```

This function uses `\cs_set_nopar:Npe` internally for performance reasons, the argument #1 is already a string in every usage, so turning it into a string again seems unnecessary. The expansion part of a key is always stored in the same place, so that is not passed along.

```

23294 \cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2#3
23295 {
23296   \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw
23297   / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
23298 }
23299 \cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
23300 {
23301   #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
23302 }
23303 \cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
23304   #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxii:Nw
23305 {
23306   \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
23307   \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
23308 }
23309 \cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
23310 {
23311   \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
23312   \__keys_find_key_module_auxi:Nw #1
23313 }
23314 \cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
23315   #1 #2 \s__keys_nil #3 \s__keys_mark
23316   \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
23317 {
23318   \exp_not:N \__keys_find_key_module_auxv:Nw #4
23319   #2 \token_to_str:N : n \token_to_str:N : \s__keys_mark
23320 }
23321 \use:e
23322 {
23323   \cs_new_protected:Npn \exp_not:N \__keys_find_key_module_auxv:Nw
23324     #1 #2 \token_to_str:N : #3 \token_to_str:N : #4 \s__keys_mark
23325 }
23326 {
23327   \cs_set_nopar:Npn #1 {#2}
23328   \cs_set_nopar:Npn \l__keys_exp_str {#3}
23329 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

23330 \cs_new_protected:Npn \__keys_set_selective:
23331 {
23332   \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
23333   {
23334     \clist_set_eq:Nc \l__keys_groups_clist
23335     { \c__keys_groups_root_str \l_keys_path_str }
23336     \__keys_check_groups:
23337   }
23338   {
23339     \bool_if:NTF \l__keys_exclude_bool

```

```

23340         \__keys_execute:
23341         \__keys_store_unused:
23342     }
23343 }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set. It is safe to use \clist_if_in:NnTF because both \l__keys_selective_clist and \l__keys_groups_clist contain the groups as strings, without leading/trailing spaces in any item, since the \lclist functions were applied to the result of applying \tl_to_str:n.

```

23344 \cs_new_protected:Npn \__keys_check_groups:
23345 {
23346     \bool_set_false:N \l__keys_tmp_bool
23347     \clist_map_inline:Nn \l__keys_selective_clist
23348     {
23349         \clist_if_in:NnT \l__keys_groups_clist {##1}
23350         {
23351             \bool_set_true:N \l__keys_tmp_bool
23352             \clist_map_break:
23353         }
23354     }
23355     \bool_if:NTF \l__keys_tmp_bool
23356     {
23357         \bool_if:NTF \l__keys_exclude_bool
23358         \__keys_store_unused:
23359         \__keys_execute:
23360     }
23361     {
23362         \bool_if:NTF \l__keys_exclude_bool
23363         \__keys_execute:
23364         \__keys_store_unused:
23365     }
23366 }

```

(End of definition for __keys_set_keyval:n and others.)

```

\__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.
\__keys_default_inherit:
\__keys_value_set:Nn
\__keys_value_set:No
\__keys_value_set:NV
\__keys_value_set:Nv
\__keys_value_set:Ne
\__keys_value_set:NN
\__keys_value_set:Nc
23367 \cs_new_protected:Npn \__keys_value_or_default:n #1
23368 {
23369     \bool_if:NTF \l__keys_no_value_bool
23370     {
23371         \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
23372         {
23373             \tl_set_eq:Nc
23374             \l_keys_value_tl
23375             { \c__keys_default_root_str \l_keys_path_str }
23376         }
23377         {
23378             \tl_clear:N \l_keys_value_tl
23379             \cs_if_exist:cT
23380             { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
23381             { \__keys_default_inherit: }
23382         }

```



```

23383     }
23384     {
23385         \cs_if_exist_use:cF { __keys_value_set:N \l__keys_exp_str }
23386         {
23387             \msg_error:nnV { keys } { unknown-expansion } \l__keys_exp_str
23388             \use_none:nn
23389         }
23390         \l_keys_value_tl {#1}
23391     }
23392 }
23393 \cs_new_protected:Npn \__keys_default_inherit:
23394 {
23395     \clist_map_inline:cn
23396     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
23397     {
23398         \cs_if_exist:cT
23399         { \c__keys_default_root_str ##1 / \l_keys_key_str }
23400         {
23401             \tl_set_eq:Nc
23402             \l_keys_value_tl
23403             { \c__keys_default_root_str ##1 / \l_keys_key_str }
23404             \clist_map_break:
23405         }
23406     }
23407 }
23408 \cs_new_eq:NN \__keys_value_set:Nn \tl_set:Nn
23409 \cs_generate_variant:Nn \__keys_value_set:Nn { No , NV , Nv , Ne }
23410 \cs_new_protected:Npn \__keys_value_set:NN #1#2 { \tl_set:Nn #1 {#2} }
23411 \cs_generate_variant:Nn \__keys_value_set:NN { Nc }

```

(End of definition for __keys_value_or_default:n and others.)

__keys_execute: Actually executing a key is done in two parts. First, look for the key itself, then look for the **unknown** key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

\__keys_execute:nn \cs_new_protected:Npn \__keys_execute:
\__keys_execute:no \cs_new_protected:Npn \__keys_execute:
\__keys_store_unused: \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
\__keys_store_unused_aux: \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
\cs_if_exist_use:c { \c__keys_check_root_str \l_keys_path_str }
\__keys_execute:no \l_keys_path_str \l_keys_value_tl
\cs_if_exist:cTF
{ \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
{ \__keys_execute_inherit: }
{ \__keys_execute_unknown: }

```

Dealing with inheritance and recursion makes life a little interesting.

```

23426 \cs_new_protected:Npn \__keys_execute_inherit:
23427 {

```

```

23428     \bool_set_false:N \l__keys_inherit_bool
23429     \clist_map_inline:cn
23430       { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
23431       { \__keys_execute_inherit:n {##1} }
23432     \bool_if:NF \l__keys_inherit_bool
23433     { \__keys_execute_unknown: }
23434   }
23435   \cs_new_protected:Npn \__keys_execute_inherit:n #1
23436   {
23437     \cs_if_exist:cTF
23438       { \c__keys_code_root_str #1 / \l_keys_key_str }
23439       {
23440         \str_set:Nn \l__keys_inherit_str {#1}
23441         \cs_if_exist_use:c { \c__keys_check_root_str #1 / \l_keys_key_str }
23442         \__keys_execute:no { #1 / \l_keys_key_str } \l_keys_value_tl
23443         \clist_map_break:n
23444         { \bool_set_true:N \l__keys_inherit_bool }
23445       }
23446     {
23447       \cs_if_exist:cT
23448         { \c__keys_inherit_root_str #1 }
23449         {
23450           \clist_map_inline:cn { \c__keys_inherit_root_str #1 }
23451           { \__keys_execute_inherit:n {##1} }
23452           \bool_if:NT \l__keys_inherit_bool
23453           { \clist_map_break: }
23454         }
23455       }
23456     }
23457   \cs_new_protected:Npn \__keys_execute_unknown:
23458   {
23459     \bool_if:NTF \l__keys_only_known_bool
23460     { \__keys_store_unused: }
23461     {
23462       \cs_if_exist:cTF
23463         { \c__keys_code_root_str \l__keys_module_str / unknown }
23464         {
23465           \bool_if:NT \l__keys_no_value_bool
23466           {
23467             \cs_if_exist:cT
23468               { \c__keys_default_root_str \l__keys_module_str / unknown }
23469               {
23470                 \tl_set_eq:Nc
23471                 \l_keys_value_tl
23472                 { \c__keys_default_root_str \l__keys_module_str / unknown }
23473               }
23474             }
23475           \__keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl
23476         }
23477       {
23478         \msg_error:nnee { keys } { unknown }
23479         \l_keys_path_str \l__keys_module_str
23480       }
23481     }

```

23482 } }

A key's code is in the control sequence with csname `\c__keys_code_root_str #1`. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```
23483 \cs_new:Npn \__keys_execute:nn #1#2
23484 { \__keys_execute:no {#1} { \prg_do_nothing: #2 } }
23485 \cs_new:Npn \__keys_execute:no #1#2
23486 {
23487   \exp_args:NNo \exp_args:No \use:n
23488   {
23489     \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
23490     \exp_after:wN {#2}
23491   }
23492 }
```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q__keys_no_value`. Then, use a standard delimited approach to fish out the partial path.

```
23493 \cs_new_protected:Npn \__keys_store_unused:
23494 {
23495   \__keys_quark_if_no_value:NTF \l__keys_relative_tl
23496   {
23497     \clist_put_right:Ne \l__keys_unused_clist
23498     {
23499       \l_keys_key_str
23500       \bool_if:NF \l__keys_no_value_bool
23501       { = { \exp_not:o \l_keys_value_tl } }
23502     }
23503   }
23504   {
23505     \tl_if_empty:NTF \l__keys_relative_tl
23506     {
23507       \clist_put_right:Ne \l__keys_unused_clist
23508       {
23509         \l_keys_path_str
23510         \bool_if:NF \l__keys_no_value_bool
23511         { = { \exp_not:o \l_keys_value_tl } }
23512       }
23513     }
23514     { \__keys_store_unused_aux: }
23515   }
23516 }
23517 \cs_new_protected:Npn \__keys_store_unused_aux:
23518 {
23519   \__kernel_tl_set:Nx \l__keys_relative_tl
23520   { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
23521   \use:e
```

```

23522     {
23523         \cs_set_protected:Npn \__keys_store_unused:w
23524             ##1 \l__keys_relative_tl /
23525             ##2 \l__keys_relative_tl /
23526             ##3 \s__keys_stop
23527     }
23528     {
23529         \tl_if_blank:nF {##1}
23530         {
23531             \msg_error:nnee { keys } { bad-relative-key-path }
23532             \l_keys_path_str
23533             \l__keys_relative_tl
23534         }
23535         \clist_put_right:Ne \l__keys_unused_clist
23536         {
23537             \exp_not:n {##2}
23538             \bool_if:NF \l__keys_no_value_bool
23539             { = { \exp_not:o \l_keys_value_tl } }
23540         }
23541     }
23542     \use:e
23543     {
23544         \__keys_store_unused:w \l_keys_path_str
23545         \l__keys_relative_tl / \l__keys_relative_tl /
23546         \s__keys_stop
23547     }
23548 }
23549 \cs_new_protected:Npn \__keys_store_unused:w { }

```

(End of definition for __keys_execute: and others.)

__keys_choice_find:n Executing a choice has two parts. First, try the choice given, then if that fails call the
 __keys_choice_find:nn unknown key. That always exists, as it is created when a choice is first made. So there
 __keys_multichoice_find:n is no need for any escape code. For multiple choices, the same code ends up used in a
 mapping.

```

23550 \cs_new:Npn \__keys_choice_find:n #1
23551     {
23552         \str_if_empty:NTF \l__keys_inherit_str
23553         { \__keys_choice_find:nn \l_keys_path_str {#1} }
23554         {
23555             \__keys_choice_find:nn
23556             { \l__keys_inherit_str / \l_keys_key_str } {#1}
23557         }
23558     }
23559 \cs_new:Npn \__keys_choice_find:nn #1#2
23560     {
23561         \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
23562         { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
23563         { \__keys_execute:nn { #1 / unknown } {#2} }
23564     }
23565 \cs_new:Npn \__keys_multichoice_find:n #1
23566     { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End of definition for __keys_choice_find:n, __keys_choice_find:nn, and __keys_multichoice_find:n.)

70.7 Utilities

```

    \__keys_parent:o      Used to strip off the ending part of the key path after the last /.
    \__keys_parent_auxi:w 23567 \cs_new:Npn \__keys_parent:o #1
    \__keys_parent_auxii:w 23568 {
    \__keys_parent_auxiii:n 23569   \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
    \__keys_parent_auxiv:w 23570   / \q_nil \__keys_parent_auxiv:w
    23571 }
    23572 \cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
    23573 {
    23574   #3 { #1 } #2 \q_nil #3
    23575 }
    23576 \cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
    23577 {
    23578   #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
    23579 }
    23580 \cs_new:Npn \__keys_parent_auxiii:n #1
    23581 {
    23582   / #1 \__keys_parent_auxi:w
    23583 }
    23584 \cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w
    23585 {
    23586 }

```

(End of definition for __keys_parent:o and others.)

```

    \__keys_trim_spaces:n Space stripping has to allow for the fact that the key here might have several parts, and
    \__keys_trim_spaces_auxi:w spaces need to be stripped from each part. Since the key name is turned into a string
    \__keys_trim_spaces_auxii:w groups can't be stripped accidentally and the precautions of \tl_trim_spaces:n aren't
    \__keys_trim_spaces_auxiii:w necessary, in this case it is much faster to just directly strip spaces around /.

```

```

23587 \group_begin:
23588   \cs_set:Npn \__keys_tmp:w #1
23589   {
23590     \cs_new:Npn \__keys_trim_spaces:n ##1
23591     {
23592       \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
23593       \s_keys_nil \__keys_trim_spaces_auxi:w
23594       \s_keys_mark \__keys_trim_spaces_auxii:w
23595       #1 / #1
23596       \s_keys_nil \__keys_trim_spaces_auxii:w
23597       \s_keys_mark \__keys_trim_spaces_auxiii:w
23598     }
23599   }
23600   \__keys_tmp:w { ~ }
23601 \group_end:
23602 \cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s_keys_nil #3
23603 {
23604   #3 #1 / #2 \s_keys_nil #3
23605 }
23606 \cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s_keys_mark #3
23607 {
23608   #3 #1 / #2 \s_keys_mark #3
23609 }
23610 \cs_new:Npn \__keys_trim_spaces_auxiii:w

```

```

23611 / #1 /
23612 \s__keys_nil \__keys_trim_spaces_auxi:w
23613 \s__keys_mark \__keys_trim_spaces_auxii:w
23614 /
23615 \s__keys_nil \__keys_trim_spaces_auxii:w
23616 \s__keys_mark \__keys_trim_spaces_auxiii:w
23617 {
23618 #1
23619 }

```

(End of definition for __keys_trim_spaces:n and others.)

\keys_if_exist_p:nn A utility for others to see if a key exists.

```

\keys_if_exist:nnTF 23620 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
\__keys_if_exist:nn 23621 {
\__keys_if_exist:ee 23622 \__keys_if_exist:ee
23623 { \__keys_trim_spaces:n {#1} }
23624 { \__keys_trim_spaces:n {#2} }
23625 }
23626 \prg_generate_conditional_variant:Nnn \keys_if_exist:nn { ne } { p , T , F , TF }
23627 \cs_new:Npn \__keys_if_exist:nn #1#2
23628 {
23629 \cs_if_exist:cTF
23630 { \c__keys_code_root_str #1 \tl_if_blank:nF {#1} { / } #2 }
23631 { \prg_return_true: }
23632 { \prg_return_false: }
23633 }
23634 \cs_generate_variant:Nn \__keys_if_exist:nn { ee }

```

(End of definition for \keys_if_exist:nnTF and __keys_if_exist:nn. This function is documented on page 258.)

\keys_if_choice_exist_p:nnn Just an alternative view on \keys_if_exist:nnTF.

```

\keys_if_choice_exist:nnnTF 23635 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
\__keys_if_exist:nnn 23636 { p , T , F , TF }
\__keys_if_exist:eee 23637 {
23638 \__keys_if_exist:eee
23639 { \__keys_trim_spaces:n {#1} }
23640 { \__keys_trim_spaces:n {#2} }
23641 { \__keys_trim_spaces:n {#3} }
23642 }
23643 \cs_new:Npn \__keys_if_exist:nnn #1#2#3
23644 {
23645 \cs_if_exist:cTF
23646 {
23647 \c__keys_code_root_str
23648 #1 \tl_if_blank:nF {#1} { / }
23649 #2 \tl_if_blank:nF {#2} { / }
23650 #3
23651 }
23652 { \prg_return_true: }
23653 { \prg_return_false: }
23654 }
23655 \cs_generate_variant:Nn \__keys_if_exist:nnn { eee }

```

(End of definition for \keys_if_choice_exist:nnnTF and _keys_if_exist:nnn. This function is documented on page 258.)

```

\keys_show:nn
\keys_log:nn
\_keys_show:Nnn
\_keys_show_aux:Nnn
\_keys_show_aux:Nee
\_keys_show:n
\_keys_show:w
\_keys_show:Nw

To show a key, show its code using a message.

23656 \cs_new_protected:Npn \keys_show:nn
23657 { \_keys_show:Nnn \msg_show:nneeee }
23658 \cs_new_protected:Npn \keys_log:nn
23659 { \_keys_show:Nnn \msg_log:nneeee }
23660 \cs_new_protected:Npn \_keys_show:Nnn #1#2#3
23661 {
23662   \_keys_show_aux:Nee
23663   #1
23664   { \_keys_trim_spaces:n {#2} }
23665   { \_keys_trim_spaces:n {#3} }
23666 }
23667 \cs_new_protected:Npn \_keys_show_aux:Nnn #1#2#3
23668 {
23669   #1 { keys } { show-key }
23670   { #2 \tl_if_blank:nF {#2} { / } #3 }
23671   {
23672     \keys_if_exist:nnT {#2} {#3}
23673     {
23674       \exp_args:Nnf \msg_show_item_unbraced:nn { code }
23675       {
23676         \exp_args:Ne \_keys_show:n
23677         {
23678           \exp_args:Nc \cs_replacement_spec:N
23679           {
23680             \c_keys_code_root_str
23681             #2 \tl_if_blank:nF {#2} { / } #3
23682           }
23683         }
23684       }
23685     }
23686   }
23687   { } { }
23688 }
23689 \cs_generate_variant:Nn \_keys_show_aux:Nnn { Nee }
23690 \cs_new:Npe \_keys_show:n #1
23691 {
23692   \exp_not:N \_keys_show:w
23693   #1
23694   \tl_to_str:n { \_keys_precompile:n }
23695   #1
23696   \tl_to_str:n { \_keys_precompile:n }
23697   \exp_not:N \s_keys_stop
23698 }
23699 \use:e
23700 {
23701   \cs_new:Npn \exp_not:N \_keys_show:w
23702   #1 \tl_to_str:n { \_keys_precompile:n }
23703   #2 \tl_to_str:n { \_keys_precompile:n }
23704   #3 \exp_not:N \s_keys_stop
23705 }

```

```

23706 {
23707   \tl_if_blank:nTF {#2}
23708     {#1}
23709     { \__keys_show:Nw #2 \s__keys_stop }
23710 }
23711 \use:e
23712 {
23713   \cs_new:Npn \exp_not:N \__keys_show:Nw #1#2
23714     \c_right_brace_str \exp_not:N \s__keys_stop
23715 }
23716 {#2}

```

(End of definition for \keys_show:nn and others. These functions are documented on page 258.)

70.8 Messages

For when there is a need to complain.

```

23717 \msg_new:nnnn { keys } { bad-relative-key-path }
23718 { The-key~'#1'~is~not~inside~the~'#2'~path. }
23719 { The-key~'#1'~cannot~be~expressed~relative~to~path~'#2'. }
23720 \msg_new:nnnn { keys } { boolean-values-only }
23721 { Key~'#1'~accepts~boolean~values~only. }
23722 { The-key~'#1'~only~accepts~the~values~'true'~and~'false'. }
23723 \msg_new:nnnn { keys } { choice-unknown }
23724 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
23725 {
23726   The-key~'#1'~only~accepts~predefined~values,~
23727   and~'#2'~is~not~one~of~these.
23728 }
23729 \msg_new:nnnn { keys } { unknown }
23730 { The-key~'#1'~is~unknown~and~is~being~ignored. }
23731 {
23732   The-module~'#2'~does~not~have~a~key~called~'#1'.\\
23733   Check~that~you~have~spelled~the~key~name~correctly.
23734 }
23735 \msg_new:nnnn { keys } { unknown-expansion }
23736 { The-value~expansion~'#1'~is~unknown. }
23737 {
23738   Key-values~can~only~be~expanded~using~one~of~the~pre-defined~methods:~
23739   n,~o,~V,~v,~e,~N~or~c.
23740 }
23741 \msg_new:nnnn { keys } { nested-choice-key }
23742 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
23743 {
23744   The-key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
23745   itself~a~choice.
23746 }
23747 \msg_new:nnnn { keys } { value-forbidden }
23748 { The-key~'#1'~does~not~take~a~value. }
23749 {
23750   The-key~'#1'~should~be~given~without~a~value.\\
23751   The-value~'#2'~was~present:~the~key~will~be~ignored.
23752 }

```



```

23753 \msg_new:nnnn { keys } { value-required }
23754 { The~key~'#1'~requires~a~value. }
23755 {
23756     The~key~'#1'~must~have~a~value.\\
23757     No~value~was~present:~the~key~will~be~ignored.
23758 }
23759 \msg_new:nnn { keys } { show-key }
23760 {
23761     The~key~#1~
23762     \tl_if_empty:nTF {#2}
23763     { is~undefined. }
23764     { has~the~properties: #2 . }
23765 }
23766 \prop_gput:Nnn \g_msg_module_name_prop { keys } { LaTeX }
23767 \prop_gput:Nnn \g_msg_module_type_prop { keys } { }
23768 \end{code}

```

Chapter 71

l3intarray implementation

23769 `<@@=intarray>`

23770 `<*code>`

There are two implementations for this module: One `\fontdimen` based one for more traditional $\TeX$ engines and a Lua based one for engines with Lua support.

Both versions do not allow negative array sizes.

23771 `\msg_new:nnn { kernel } { negative-array-size }`

23772 `{ Size-of-array-may-not-be-negative:~#1 }`

`\__intarray_sep:`

23773 `\cs_new_eq:NN \__intarray_sep: \__kernel_int_sep:`

(End of definition for __intarray_sep:.)

`\l__intarray_loop_int` A loop index.

23774 `\int_new:N \l__intarray_loop_int`

(End of definition for \l__intarray_loop_int.)

71.1 Lua implementation

First, let's look at the Lua variant:

We select the Lua version if the Lua helpers were defined. This can be detected by the presence of `\__intarray_gset_count:Nw`.

23775 `\cs_if_exist:NTF \__intarray_gset_count:Nw`

23776 `{`

71.1.1 Allocating arrays

`\g__intarray_table_int` Used to differentiate intarrays in Lua and to record an invalid index.
`\l__intarray_bad_index_int`

23777 `\int_new:N \g__intarray_table_int`

23778 `\int_new:N \l__intarray_bad_index_int`

23779 `</code>`

(End of definition for \g__intarray_table_int and \l__intarray_bad_index_int.)

`\_intarray:w` Used as marker for intarrays in Lua. Followed by an unbraced number identifying the array and a single space. This format is used to make it easy to scan from Lua.

```

23780 (*lua)
23781 luacmd('\_intarray:w', function()
23782     scan_int()
23783     tex.error'LaTeX Error: Isolated intarray ignored'
23784 end, 'protected', 'global')
23785 (/lua)

```

(End of definition for `\_intarray:w`.)

`\intarray_new:Nn` Declare #1 as a tokenlist with the scanmark and a unique number. Pass the array's size to the Lua helper. Every intarray must be global; it's enough to run this check in `\intarray_new:Nn`.

```

23786 (*code)
23787 \cs_new_protected:Npn \_intarray_new:N #1
23788 {
23789     \_kernel_chk_if_free_cs:N #1
23790     \int_gincr:N \g\_intarray_table_int
23791     \cs_gset_nopar:Npe #1 { \_intarray:w \int_use:N \g\_intarray_table_int \c_space_tl
23792 }
23793 \cs_new_protected:Npn \intarray_new:Nn #1#2
23794 {
23795     \_intarray_new:N #1
23796     \_intarray_gset_count:Nw #1 \int_eval:n {#2} \scan_stop:
23797     \int_compare:nNnT { \intarray_count:N #1 } < 0
23798     {
23799         \msg_error:nne { kernel } { negative-array-size }
23800         { \intarray_count:N #1 }
23801     }
23802 }
23803 \cs_generate_variant:Nn \intarray_new:Nn { c }
23804 (/code)

```

(End of definition for `\intarray_new:Nn` and `\_intarray_new:N`. This function is documented on page 262.)

Before we get to the first command implemented in Lua, we first need some definitions. Since `token.create` only works correctly if `TEX` has seen the tokens before, we first run a short `TEX` sequence to ensure that all relevant control sequences are known.

```

23805 (*lua)
23806
23807 local scan_token = token.scan_token
23808 local put_next = token.put_next
23809 local intarray_marker = token_create_safe'\_intarray:w'
23810 local use_none = token_create_safe'use_none:n'
23811 local use_i = token_create_safe'use:n'
23812 local expand_after_scan_stop = {token_create_safe'exp_after:wN',
23813                                 token_create_safe'scan_stop:'}
23814 local comma = token_create(string.byte',')

```

`\_intarray_table` Internal helper to scan an intarray token, extract the associated Lua table and return an error if the input is invalid.

```

23815 local \_intarray_table do

```

```

23816 local tables = get_luadata and get_luadata'__intarray' or {[0] = {}}
23817 function __intarray_table()
23818   local t = scan_token()
23819   if t ~= intarray_marker then
23820     put_next(t)
23821     tex.error'LaTeX Error: intarray expected'
23822     return tables[0]
23823   end
23824   local i = scan_int()
23825   local current_table = tables[i]
23826   if current_table then return current_table end
23827   current_table = {}
23828   tables[i] = current_table
23829   return current_table
23830 end

```

Since in L^AT_EX this is loaded in the format, we want to preserve any intarrays which are created while format building for the actual run.

To do this, we use the `register_luadata` mechanism from l3luatex: Directly before the format get dumped, the following function gets invoked and serializes all existing tables into a string. This string gets compiled and dumped into the format and is made available at the beginning of regular runs as `get_luadata'@@'`.

```

23831 if register_luadata then
23832   register_luadata('__intarray', function()
23833     local t = "{[0]={},"
23834     for i=1, #tables do
23835       t = string.format("%s{%s},"", t, table.concat(tables[i], ','))
23836     end
23837     return t .. "}"
23838   end)
23839 end
23840 end

```

(End of definition for __intarray_table.)

`\intarray_count:N` Set and get the size of an array. “Setting the size” means in this context that we add zeros until we reach the desired size.
`\intarray_count:c`
`\__intarray_gset_count:Nw`

```

23841
23842 local sprint = tex.sprint
23843
23844 luacmd('__intarray_gset_count:Nw', function()
23845   local t = __intarray_table()
23846   local n = scan_int()
23847   for i=#t+1, n do t[i] = 0 end
23848 end, 'protected', 'global')
23849
23850 luacmd('intarray_count:N', function()
23851   sprint(-2, #__intarray_table())
23852 end, 'global')
23853 </lua>
23854 <code>
23855   \cs_generate_variant:Nn \intarray_count:N { c }
23856 </code>

```

(End of definition for `\intarray_count:N` and `\__intarray_gset_count:Nw`. This function is documented on page 263.)

71.1.2 Array items

`\__intarray_gset:wF` The setter provided by Lua. The argument order somewhat emulates the `\fontdimen:`
`\__intarray_gset:w` First the array index, followed by the intarray and then the new value. This has been chosen over a more conventional order to provide a delimiter for the numbers.

```

23857 (*lua)
23858 luacmd('\__intarray_gset:wF', function()
23859   local i = scan_int()
23860   local t = __intarray_table()
23861   if t[i] then
23862     t[i] = scan_int()
23863     put_next(use_none)
23864   else
23865     tex.count.l__intarray_bad_index_int = i
23866     scan_int()
23867     put_next(use_i)
23868   end
23869 end, 'protected', 'global')
23870
23871 luacmd('\__intarray_gset:w', function()
23872   local i = scan_int()
23873   local t = __intarray_table()
23874   t[i] = scan_int()
23875 end, 'protected', 'global')
23876 
```

(End of definition for `\__intarray_gset:wF` and `\__intarray_gset:w`.)

`\intarray_gset:Nnn` The `\__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its
`\intarray_gset:cnn` arguments must be suitable for `\int_value:w`. The user version checks the position and
`\__kernel_intarray_gset:Nnn` value are within bounds.

```

23877 (*code)
23878 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
23879 { \__intarray_gset:w #2 #1 #3 \scan_stop: }
23880 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
23881 {
23882   \__intarray_gset:wF \int_eval:n {#2} #1 \int_eval:n{#3}
23883   {
23884     \msg_error:nneee { kernel } { out-of-bounds }
23885     { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
23886   }
23887 }
23888 \cs_generate_variant:Nn \intarray_gset:Nnn { c }
23889 
```

(End of definition for `\intarray_gset:Nnn` and `\__kernel_intarray_gset:Nnn`. This function is documented on page 263.)

`\intarray_gzero:N` Set the appropriate array entry to zero. No bound checking needed.
`\intarray_gzero:c`

```

23890 (*lua)
23891 luacmd('intarray_gzero:N', function()

```

```

23892 local t = __intarray_table()
23893 for i=1, #t do
23894     t[i] = 0
23895 end
23896 end, 'global', 'protected')
23897  $\langle$ /lua $\rangle$ 
23898  $\langle$ *code $\rangle$ 
23899     \cs_generate_variant:Nn \intarray_gzero:N { c }
23900  $\langle$ /code $\rangle$ 

```

(End of definition for \intarray_gzero:N. This function is documented on page 262.)

```

\intarray_item:Nn Get the appropriate entry and perform bound checks. The \__kernel_intarray_
\intarray_item:cn item:Nn function omits bound checks and omits \int_eval:n, namely its argument
\__kernel_intarray_item:Nn must be a TEX integer suitable for \int_value:w.
\__intarray_item:wF
\__intarray_item:w
23901  $\langle$ *lua $\rangle$ 
23902 luacmd('__intarray_item:wF', function()
23903     local i = scan_int()
23904     local t = __intarray_table()
23905     local item = t[i]
23906     if item then
23907         put_next(use_none)
23908     else
23909         tex.l__intarray_bad_index_int = i
23910         put_next(use_i)
23911     end
23912     put_next(expand_after_scan_stop)
23913     scan_token()
23914     if item then
23915         sprint(-2, item)
23916     end
23917 end, 'global')
23918
23919 luacmd('__intarray_item:w', function()
23920     local i = scan_int()
23921     local t = __intarray_table()
23922     sprint(-2, t[i])
23923 end, 'global')
23924  $\langle$ /lua $\rangle$ 
23925  $\langle$ *code $\rangle$ 
23926     \cs_new:Npn \__kernel_intarray_item:Nn #1#2
23927         { \__intarray_item:w #2 #1 }
23928     \cs_new:Npn \intarray_item:Nn #1#2
23929         {
23930             \__intarray_item:wF \int_eval:n {#2} #1
23931             {
23932                 \msg_expandable_error:nnfff { kernel } { out-of-bounds }
23933                 { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
23934                     0
23935             }
23936         }
23937     \cs_generate_variant:Nn \intarray_item:Nn { c }

```

(End of definition for \intarray_item:Nn and others. This function is documented on page 263.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c
23938 \cs_new:Npn \intarray_rand_item:N #1
23939 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
23940 \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End of definition for `\intarray_rand_item:N`. This function is documented on page 263.)

71.1.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` We use the `\__kernel_intarray_gset:Nnn` which does not do bounds checking and instead automatically resizes the array. This is not implemented in Lua to ensure that the clist parsing is consistent with the clist module.

```

\intarray_const_from_clist:cn
23941 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
23942 {
23943   \__intarray_new:N #1
23944   \int_zero:N \l__intarray_loop_int
23945   \clist_map_inline:nn {#2}
23946   {
23947     \int_incr:N \l__intarray_loop_int
23948     \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int { \int_eval:n {##1} } }
23949   }
23950   \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }

```

(End of definition for `\intarray_const_from_clist:Nn`. This function is documented on page 262.)

`\__intarray_to_clist:Nn` The `\__intarray_to_clist:Nn` auxiliary allows to choose the delimiter and is also used in `\intarray_show:N`. Here we just pass the information to Lua and let `table.concat` do the actual work. We discard the category codes of the passed delimiter but this is not an issue since the delimiter is always just a comma or a comma and a space. In both cases `sprint(2, ...)` provides the right catcodes.

`\__intarray_to_clist:w`

```

23951 \code
23952 \lua
23953 local concat = table.concat
23954 luacmd('\__intarray_to_clist:Nn', function()
23955   local t = __intarray_table()
23956   local sep = token.scan_string()
23957   sprint(-2, concat(t, sep))
23958 end, 'global')
23959 \lua

```

(End of definition for `\__intarray_to_clist:Nn` and `\__intarray_to_clist:w`.)

Loop through part of the array.

```

\__kernel_intarray_range_to_clist:Nnn
\__intarray_range_to_clist:w
23960 \code
23961 \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
23962 {
23963   \__intarray_range_to_clist:w #1
23964   \int_eval:n {#2} ~ \int_eval:n {#3} ~
23965 }
23966 \code
23967 \lua
23968 luacmd('\__intarray_range_to_clist:w', function()
23969   local t = __intarray_table()
23970   local from = scan_int()

```

```

23971   local to = scan_int()
23972   sprint(-2, concat(t, ', ', from, to))
23973 end, 'global')
23974  $\langle$ /lua $\rangle$ 

```

(End of definition for `\_kernel_intarray_range_to_clist:Nnn` and `\_intarray_range_to_clist:w`.)

`\_kernel_intarray_gset_range_from_clist:Nnn`
`\_intarray_gset_range:nNw`

Loop through part of the array. We allow additional commas at the end.

```

23975  $\langle$ *code $\rangle$ 
23976   \cs_new_protected:Npn \_kernel_intarray_gset_range_from_clist:Nnn #1#2#3
23977   {
23978     \_intarray_gset_range:w \int_eval:w #2 #1 #3 , , \scan_stop:
23979   }
23980  $\langle$ /code $\rangle$ 
23981  $\langle$ *lua $\rangle$ 
23982 luacmd('\_intarray_gset_range:w', function()
23983   local from = scan_int()
23984   local t = \_intarray_table()
23985   while true do
23986     local tok = scan_token()
23987     if tok == comma then
23988       repeat
23989         tok = scan_token()
23990       until tok ~= comma
23991       break
23992     else
23993       put_next(tok)
23994     end
23995     t[from] = scan_int()
23996     scan_token()
23997     from = from + 1
23998   end
23999 end, 'global', 'protected')
24000  $\langle$ /lua $\rangle$ 

```

(End of definition for `\_kernel_intarray_gset_range_from_clist:Nnn` and `\_intarray_gset_range:nNw`.)

`\_intarray_gset_overflow_test:nw`

In order to allow some code sharing later we provide the `\_intarray_gset_overflow_test:nw` name here. It doesn't actually test anything since the Lua implementation accepts all integers which could be tested with `\tex_ifabsnum:D`.

```

24001  $\langle$ *code $\rangle$ 
24002   \cs_new_protected:Npn \_intarray_gset_overflow_test:nw #1
24003   {
24004   }

```

(End of definition for `\_intarray_gset_overflow_test:nw`.)

71.2 Font dimension based implementation

Go to the false branch of the conditional above.

```

24005   }
24006   {

```


71.2.1 Allocating arrays

<code>__intarray_entry:w</code>	We use these primitives quite a lot in this module.
<code>__intarray_count:w</code>	<pre> 24007 \cs_new_eq:NN __intarray_entry:w \tex_fontdimen:D 24008 \cs_new_eq:NN __intarray_count:w \tex_hyphenchar:D </pre> (End of definition for <code>__intarray_entry:w</code> and <code>__intarray_count:w</code>.)
<code>\c__intarray_sp_dim</code>	Used to convert integers to dimensions fast. <pre> 24009 \dim_const:Nn \c__intarray_sp_dim { 1 sp } </pre> (End of definition for <code>\c__intarray_sp_dim</code>.)
<code>\g__intarray_font_int</code>	Used to assign one font per array. <pre> 24010 \int_new:N \g__intarray_font_int </pre> (End of definition for <code>\g__intarray_font_int</code>.)
<code>\intarray_new:Nn</code>	Declare <code>#1</code> to be a font (arbitrarily <code>cmr10</code> at a never-used size). Store the array's size as
<code>\intarray_new:cn</code>	the <code>\hyphenchar</code> of that font and make sure enough <code>\fontdimen</code> are allocated, by setting
<code>__intarray_new:N</code>	the last one. Then clear any <code>\fontdimen</code> that <code>cmr10</code> starts with. It seems LuaTeX's <code>cmr10</code> has an extra <code>\fontdimen</code> parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every <code>intarray</code> must be global; it's enough to run this check in <code>\intarray_new:Nn</code> .
	<pre> 24011 \cs_new_protected:Npn __intarray_new:N #1 24012 { 24013 __kernel_chk_if_free_cs:N #1 24014 \int_gincr:N \g__intarray_font_int 24015 \tex_global:D \tex_font:D #1 24016 = cmr10~at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop: 24017 \int_step_inline:nn { 8 } 24018 { __kernel_intarray_gset:Nnn #1 {##1} \c_zero_int } 24019 } 24020 \cs_new_protected:Npn \intarray_new:Nn #1#2 24021 { 24022 __intarray_new:N #1 24023 __intarray_count:w #1 = \int_eval:n {#2} \scan_stop: 24024 \int_compare:nNnT { \intarray_count:N #1 } < 0 24025 { 24026 \msg_error:nne { kernel } { negative-array-size } 24027 { \intarray_count:N #1 } 24028 } 24029 \int_compare:nNnT { \intarray_count:N #1 } > 0 24030 { __kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } } 24031 } 24032 \cs_generate_variant:Nn \intarray_new:Nn { c } </pre> (End of definition for <code>\intarray_new:Nn</code> and <code>__intarray_new:N</code>. This function is documented on page 262.)
<code>\intarray_count:N</code>	Size of an array.
<code>\intarray_count:c</code>	<pre> 24033 \cs_new:Npn \intarray_count:N #1 { \int_value:w __intarray_count:w #1 } 24034 \cs_generate_variant:Nn \intarray_count:N { c } </pre> (End of definition for <code>\intarray_count:N</code>. This function is documented on page 263.)

71.2.2 Array items

`\__intarray_signed_max_dim:n` Used when an item to be stored is larger than `\c_max_dim` in absolute value; it is replaced by $\pm\c\_max\_dim$.

```
24035 \cs_new:Npn \__intarray_signed_max_dim:n #1
24036 { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End of definition for `\__intarray_signed_max_dim:n`.)

`\__intarray_bounds:NNnTF` The functions `\intarray_gset:Nnn` and `\intarray_item:Nn` share bounds checking. `\__intarray_bounds_error:NNnw` The T branch is used if #3 is within bounds of the array #2.

```
24037 \cs_new:Npn \__intarray_bounds:NNnTF #1#2#3
24038 {
24039   \if_int_compare:w 1 > #3 \exp_stop_f:
24040     \__intarray_bounds_error:NNnw #1 #2 {#3}
24041   \else:
24042     \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
24043     \__intarray_bounds_error:NNnw #1 #2 {#3}
24044   \fi:
24045   \fi:
24046   \use_i:nn
24047 }
24048 \cs_new:Npn \__intarray_bounds_error:NNnw #1#2#3#4 \use_i:nn #5#6
24049 {
24050   #4
24051   #1 { kernel } { out-of-bounds }
24052   { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
24053   #6
24054 }
```

(End of definition for `\__intarray_bounds:NNnTF` and `\__intarray_bounds_error:NNnw`.)

`\intarray_gset:Nnn` Set the appropriate `\fontdimen`. The `\__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

`\intarray_gset:cnm`

`\__kernel_intarray_gset:Nnn`

`\__intarray_gset:Nnn`

`\__intarray_gset_overflow:Nnn`

```
24055 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
24056 { \__intarray_entry:w #2 #1 #3 \c__intarray_sp_dim }
24057 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
24058 {
24059   \exp_after:wN \__intarray_gset:Nww
24060   \exp_after:wN #1
24061   \int_value:w \int_eval:n {#2} \exp_after:wN \__intarray_sep:
24062   \int_value:w \int_eval:n {#3} \__intarray_sep:
24063 }
24064 \cs_generate_variant:Nn \intarray_gset:Nnn { c }
24065 \cs_new_protected:Npn \__intarray_gset:Nww #1#2 \__intarray_sep: #3 \__intarray_sep:
24066 {
24067   \__intarray_bounds:NNnTF \msg_error:nneee #1 {#2}
24068   {
24069     \__intarray_gset_overflow_test:nw {#3}
24070     \__kernel_intarray_gset:Nnn #1 {#2} {#3}
24071   }
24072   { }
24073 }
```

```

24074 \cs_if_exist:NTF \tex_ifabsnum:D
24075 {
24076   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
24077   {
24078     \tex_ifabsnum:D #1 > \c_max_dim
24079     \exp_after:wN \__intarray_gset_overflow:NNnn
24080     \fi:
24081   }
24082 }
24083 {
24084   \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
24085   {
24086     \if_int_compare:w \int_abs:n {#1} > \c_max_dim
24087     \exp_after:wN \__intarray_gset_overflow:NNnn
24088     \fi:
24089   }
24090 }
24091 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
24092 {
24093   \msg_error:nneeee { kernel } { overflow }
24094   { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
24095   #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
24096 }

```

(End of definition for `\intarray_gset:Nnn` and others. This function is documented on page 263.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

`\intarray_gzero:c`

```

24097 \cs_new_protected:Npn \intarray_gzero:N #1
24098 {
24099   \int_zero:N \l__intarray_loop_int
24100   \prg_replicate:nn { \intarray_count:N #1 }
24101   {
24102     \int_incr:N \l__intarray_loop_int
24103     \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
24104   }
24105 }
24106 \cs_generate_variant:Nn \intarray_gzero:N { c }

```

(End of definition for `\intarray_gzero:N`. This function is documented on page 262.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `\__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

`\intarray_item:cn`

`\__kernel_intarray_item:Nn`

`\__intarray_item:Nw`

```

24107 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
24108 { \int_value:w \__intarray_entry:w #2 #1 }
24109 \cs_new:Npn \intarray_item:Nn #1#2
24110 {
24111   \exp_after:wN \__intarray_item:Nw
24112   \exp_after:wN #1
24113   \int_value:w \int_eval:n {#2} \__intarray_sep:
24114 }
24115 \cs_generate_variant:Nn \intarray_item:Nn { c }

```

```

24116 \cs_new:Npn \__intarray_item:Nw #1#2 \__intarray_sep:
24117 {
24118   \__intarray_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
24119   { \__kernel_intarray_item:Nn #1 {#2} }
24120   { 0 }
24121 }

```

(End of definition for `\intarray_item:Nn`, `\__kernel_intarray_item:Nn`, and `\__intarray_item:Nw`. This function is documented on page 263.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c 24122 \cs_new:Npn \intarray_rand_item:N #1
24123 { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
24124 \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End of definition for `\intarray_rand_item:N`. This function is documented on page 263.)

71.2.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn` Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that `TeX` allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `\__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

24125 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
24126 {
24127   \__intarray_new:N #1
24128   \int_zero:N \l__intarray_loop_int
24129   \clist_map_inline:nn {#2}
24130   { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
24131   \__intarray_count:w #1 \l__intarray_loop_int
24132 }
24133 \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
24134 \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
24135 {
24136   \int_incr:N \l__intarray_loop_int
24137   \__intarray_gset_overflow_test:nw {#1}
24138   \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
24139 }

```

(End of definition for `\intarray_const_from_clist:Nn` and `\__intarray_const_from_clist:nN`. This function is documented on page 262.)

`\__intarray_to_clist:Nn` Loop through the array, putting a comma before each item. Remove the leading comma with `f`-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```

24140 \cs_new:Npn \__intarray_to_clist:Nn #1#2
24141 {
24142   \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
24143   {
24144     \exp_last_unbraced:Nf \use_none:n
24145     { \__intarray_to_clist:w 1 \__intarray_sep: #1 {#2} \prg_break_point: }
24146   }

```

```

24147     }
24148 \cs_new:Npn \__intarray_to_clist:w #1 \__intarray_sep: #2#3
24149 {
24150     \if_int_compare:w #1 > \__intarray_count:w #2
24151     \prg_break:n
24152     \fi:
24153     #3 \__kernel_intarray_item:Nn #2 {#1}
24154     \exp_after:wN \__intarray_to_clist:w
24155     \int_value:w \int_eval:w #1 + \c_one_int \__intarray_sep: #2 {#3}
24156 }

```

(End of definition for __intarray_to_clist:Nn and __intarray_to_clist:w.)

__kernel_intarray_range_to_clist:Nnn Loop through part of the array.

```

\__intarray_range_to_clist:ww 24157 \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
24158 {
24159     \exp_last_unbraced:Nf \use_none:n
24160     {
24161         \exp_after:wN \__intarray_range_to_clist:ww
24162         \int_value:w \int_eval:w #2 \exp_after:wN \__intarray_sep:
24163         \int_value:w \int_eval:w #3 \__intarray_sep:
24164         #1 \prg_break_point:
24165     }
24166 }
24167 \cs_new:Npn \__intarray_range_to_clist:ww #1 \__intarray_sep: #2 \__intarray_sep: #3
24168 {
24169     \if_int_compare:w #1 > #2 \exp_stop_f:
24170     \prg_break:n
24171     \fi:
24172     , \__kernel_intarray_item:Nn #3 {#1}
24173     \exp_after:wN \__intarray_range_to_clist:ww
24174     \int_value:w \int_eval:w #1 + \c_one_int \__intarray_sep: #2 \__intarray_sep: #3
24175 }

```

(End of definition for __kernel_intarray_range_to_clist:Nnn and __intarray_range_to_clist:ww.)

__kernel_intarray_gset_range_from_clist:Nnn Loop through part of the array.

```

\__intarray_gset_range:Nw 24176 \cs_new_protected:Npn \__kernel_intarray_gset_range_from_clist:Nnn #1#2#3
24177 {
24178     \int_set:Nn \l__intarray_loop_int {#2}
24179     \__intarray_gset_range:Nw #1 #3 , , \prg_break_point:
24180 }
24181 \cs_new_protected:Npn \__intarray_gset_range:Nw #1 #2 ,
24182 {
24183     \if_catcode:w \scan_stop: \tl_to_str:n {#2} \scan_stop:
24184     \prg_break:n
24185     \fi:
24186     \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
24187     \int_incr:N \l__intarray_loop_int
24188     \__intarray_gset_range:Nw #1
24189 }

```

(End of definition for __kernel_intarray_gset_range_from_clist:Nnn and __intarray_gset_range:Nw.)

```

24190 }

```

71.3 Common parts

`\intarray_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```
\intarray_if_exist_p:c 24191 \prg_new_eq_conditional:NNn \intarray_if_exist:N \cs_if_exist:N
\intarray_if_exist:NTF 24192 { TF , T , F , p }
\intarray_if_exist:cTF 24193 \prg_new_eq_conditional:NNn \intarray_if_exist:c \cs_if_exist:c
24194 { TF , T , F , p }
```

(End of definition for `\intarray_if_exist:N`. This function is documented on page 263.)

`\intarray_show:N` Convert the list to a comma list (with spaces after each comma)

```
\intarray_show:c 24195 \cs_new_protected:Npn \intarray_show:N { \__intarray_show:NN \msg_show:nneeee }
\intarray_log:N 24196 \cs_generate_variant:Nn \intarray_show:N { c }
\intarray_log:c 24197 \cs_new_protected:Npn \intarray_log:N { \__intarray_show:NN \msg_log:nneeee }
24198 \cs_generate_variant:Nn \intarray_log:N { c }
24199 \cs_new_protected:Npn \__intarray_show:NN #1#2
24200 {
24201   \__kernel_chk_defined:NT #2
24202   {
24203     #1 { intarray } { show }
24204     { \token_to_str:N #2 }
24205     { \intarray_count:N #2 }
24206     { >~ \__intarray_to_clist:Nn #2 { , ~ } }
24207     { }
24208   }
24209 }
```

(End of definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 263.)

24210 `\code`

Chapter 72

l3fp implementation

Nothing to see here: everything is in the subfiles!

Chapter 73

l3fp-aux implementation

24211 $\langle *code \rangle$

24212 $\langle @@=fp \rangle$

73.1 Access to primitives

`\__fp_int_eval:w` Largely for performance reasons, we need to directly access primitives rather than use
`\__fp_int_eval_end:` `\int_eval:n`. This happens *a lot*, so we use private names. The same is true for
`\__fp_int_to_roman:w` `\romannumeral`, although it is used much less widely.

24213 `\cs_new_eq:NN \__fp_int_eval:w \tex_numexpr:D`

24214 `\cs_new_eq:NN \__fp_int_eval_end: \scan_stop:`

24215 `\cs_new_eq:NN \__fp_int_to_roman:w \tex_romannumeral:D`

(End of definition for `\__fp_int_eval:w`, `\__fp_int_eval_end:`, and `\__fp_int_to_roman:w`.)

`\__fp_sep:`

24216 `\cs_new_eq:NN \__fp_sep: \__kernel_int_sep:`

(End of definition for `\__fp_sep:`.)

73.2 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

`\s__fp \__fp_chk:w  $\langle case \rangle$   $\langle sign \rangle$   $\langle body \rangle$  \__fp_sep:`

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to **f**-expansion. They must leave a recognizable mark after **f**-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under **e/x**-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their $\langle case \rangle$, which is a single digit:

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s__fp... __fp_sep:	Positive zero.
0 2 \s__fp... __fp_sep:	Negative zero.
1 0 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} __fp_sep:	Positive floating point.
1 2 {<exponent>} {<X ₁ >} {<X ₂ >} {<X ₃ >} {<X ₄ >} __fp_sep:	Negative floating point.
2 0 \s__fp... __fp_sep:	Positive infinity.
2 2 \s__fp... __fp_sep:	Negative infinity.
3 1 \s__fp... __fp_sep:	Quiet nan.
3 1 \s__fp... __fp_sep:	Signaling nan.

0 zeros: +0 and -0,

1 “normal” numbers (positive and negative),

2 infinities: +inf and -inf,

3 quiet and signaling nan.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of nan, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

\s__fp __fp_chk:w <case> <sign> \s__fp... __fp_sep:

where \s__fp... is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

\s__fp __fp_chk:w 1 <sign> {<exponent>} {<X₁>} {<X₂>} {<X₃>} {<X₄>} __fp_sep:

Here, the $\langle exponent \rangle$ is an integer, between -10000 and 10000 . The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the $\langle exponent \rangle$ is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, i.e. one myriad.

73.3 Using arguments and __fp_sep:s

__fp_use_none_stop_f:n This function removes an argument (typically a digit) and replaces it by \exp_stop_f:, a marker which stops f-type expansion.

24217 \cs_new:Npn __fp_use_none_stop_f:n #1 { \exp_stop_f: }

(End of definition for __fp_use_none_stop_f:n.)

`\__fp_use_s:n` Those functions place a `\__fp_sep:` after one or two arguments (typically digits).

`\__fp_use_s:nn`

```

24218 \cs_new:Npn \__fp_use_s:n #1 { #1\__fp_sep: }
24219 \cs_new:Npn \__fp_use_s:nn #1#2 { #1#2\__fp_sep: }

```

(End of definition for `\__fp_use_s:n` and `\__fp_use_s:nn`.)

`\__fp_use_none_until_s:w` Those functions select specific arguments among a set of arguments delimited by a `\__fp_sep:`.

`\__fp_use_i_until_s:nw`

`\__fp_use_ii_until_s:nnw`

```

24220 \cs_new:Npn \__fp_use_none_until_s:w #1\__fp_sep: { }
24221 \cs_new:Npn \__fp_use_i_until_s:nw #1#2\__fp_sep: { #1 }
24222 \cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3\__fp_sep: { #2 }

```

(End of definition for `\__fp_use_none_until_s:w`, `\__fp_use_i_until_s:nw`, and `\__fp_use_ii_until_s:nnw`.)

`\__fp_reverse_args:Nww` Many internal functions take arguments delimited by `\__fp_sep:s`, and it is occasionally useful to swap two such arguments.

```

24223 \cs_new:Npn \__fp_reverse_args:Nww #1 #2\__fp_sep: #3\__fp_sep:
24224 { #1 #3\__fp_sep: #2\__fp_sep: }

```

(End of definition for `\__fp_reverse_args:Nww`.)

`\__fp_rrot:www` Rotate three arguments delimited by `\__fp_sep:s`. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```

24225 \cs_new:Npn \__fp_rrot:www #1\__fp_sep: #2\__fp_sep: #3\__fp_sep:
24226 { #2\__fp_sep: #3\__fp_sep: #1\__fp_sep: }

```

(End of definition for `\__fp_rrot:www`.)

`\__fp_use_i:ww` Many internal functions take arguments delimited by `\__fp_sep:s`, and it is occasionally useful to remove one or two such arguments.

`\__fp_use_i:www`

```

24227 \cs_new:Npn \__fp_use_i:ww #1\__fp_sep: #2\__fp_sep: { #1\__fp_sep: }
24228 \cs_new:Npn \__fp_use_i:www #1\__fp_sep: #2\__fp_sep: #3\__fp_sep: { #1\__fp_sep: }

```

(End of definition for `\__fp_use_i:ww` and `\__fp_use_i:www`.)

73.4 Constants, and structure of floating points

`\__fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach T_EX's stomach.

```

24229 \cs_new_protected:Npn \__fp_misused:n #1
24230 { \msg_error:nne { fp } { misused } { \fp_to_tl:n { #1 } } }

```

(End of definition for `\__fp_misused:n`.)

`\s__fp` Floating points numbers all start with `\s__fp \__fp_chk:w`, where `\s__fp` is equal to the T_EX primitive `\relax`, and `\__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `e/x`-expansion. However, when typeset, `\s__fp` does nothing, and `\__fp_chk:w` is expanded. We define `\__fp_chk:w` to produce an error.

`\__fp_chk:w`

```

24231 \scan_new:N \s__fp
24232 \cs_new_protected:Npn \__fp_chk:w #1 \__fp_sep:
24233 { \__fp_misused:n { \s__fp \__fp_chk:w #1 \__fp_sep: } }

```

(End of definition for `\s__fp` and `\__fp_chk:w`.)

`\s__fp_expr_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s__fp_expr_stop` 24234 `\scan_new:N \s__fp_expr_mark`
24235 `\scan_new:N \s__fp_expr_stop`

(End of definition for `\s__fp_expr_mark` and `\s__fp_expr_stop`.)

`\s__fp_mark` Generic scan marks used throughout the module.

`\s__fp_stop` 24236 `\scan_new:N \s__fp_mark`
24237 `\scan_new:N \s__fp_stop`

(End of definition for `\s__fp_mark` and `\s__fp_stop`.)

`\__fp_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

24238 `\cs_new:Npn \__fp_use_i_delimit_by_s_stop:nw #1 #2 \s__fp_stop {#1}`

(End of definition for `\__fp_use_i_delimit_by_s_stop:nw`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s__fp_underflow` 24239 `\scan_new:N \s__fp_invalid`
`\s__fp_overflow` 24240 `\scan_new:N \s__fp_underflow`
`\s__fp_division` 24241 `\scan_new:N \s__fp_overflow`
`\s__fp_exact` 24242 `\scan_new:N \s__fp_division`
24243 `\scan_new:N \s__fp_exact`

(End of definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.

`\c_minus_zero_fp` 24244 `\tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact \__fp_sep: }`
`\c_inf_fp` 24245 `\tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact \__fp_sep: }`
`\c_minus_inf_fp` 24246 `\tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact \__fp_sep: }`
`\c_nan_fp` 24247 `\tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact \__fp_sep: }`
24248 `\tl_const:Nn \c_nan_fp { \s__fp \__fp_chk:w 3 1 \s__fp_exact \__fp_sep: }`

(End of definition for `\c_zero_fp` and others. These variables are documented on page 276.)

`\c__fp_prec_int` The number of digits of floating points.

`\c__fp_half_prec_int` 24249 `\int_const:Nn \c__fp_prec_int { 16 }`
`\c__fp_block_int` 24250 `\int_const:Nn \c__fp_half_prec_int { 8 }`
24251 `\int_const:Nn \c__fp_block_int { 4 }`

(End of definition for `\c__fp_prec_int`, `\c__fp_half_prec_int`, and `\c__fp_block_int`.)

`\c__fp_myriad_int` Blocks have 4 digits so this integer is useful.

24252 `\int_const:Nn \c__fp_myriad_int { 10000 }`

(End of definition for `\c__fp_myriad_int`.)

`\c__fp_minus_min_exponent_int` Normal floating point numbers have an exponent between `– minus_min_exponent` and
`\c__fp_max_exponent_int` `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0 . It would be more natural to define a `min_exponent` with the opposite sign but that would waste one TeX count.

24253 `\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }`
24254 `\int_const:Nn \c__fp_max_exponent_int { 10000 }`

(End of definition for `\c__fp_minus_min_exponent_int` and `\c__fp_max_exponent_int`.)

`\c__fp_max_exp_exponent_int` If a number's exponent is larger than that, its exponential overflows/underflows.

```

24255 \int_const:Nn \c__fp_max_exp_exponent_int { 5 }

```

(End of definition for `\c__fp_max_exp_exponent_int`.)

`\c__fp_overflowing_fp` A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```

24256 \tl_const:Nc \c__fp_overflowing_fp
24257 {
24258   \s__fp \__fp_chk:w 1 0
24259   { \int_eval:n { \c__fp_max_exponent_int + 1 } }
24260   {1000} {0000} {0000} {0000} \__fp_sep:
24261 }

```

(End of definition for `\c__fp_overflowing_fp`.)

`\__fp_zero_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

`\__fp_inf_fp:N`

```

24262 \cs_new:Npn \__fp_zero_fp:N #1
24263 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow \__fp_sep: }
24264 \cs_new:Npn \__fp_inf_fp:N #1
24265 { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow \__fp_sep: }

```

(End of definition for `\__fp_zero_fp:N` and `\__fp_inf_fp:N`.)

`\__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0. This is used in `l3str-format`.

```

24266 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
24267 {
24268   \if_meaning:w 1 #1
24269     \exp_after:wN \__fp_use_ii_until_s:nnw
24270   \else:
24271     \exp_after:wN \__fp_use_i_until_s:nw
24272     \exp_after:wN 0
24273   \fi:
24274 }

```

(End of definition for `\__fp_exponent:w`.)

`\__fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (nan) to 1, and 2 to 0.

```

24275 \cs_new:Npn \__fp_neg_sign:N #1
24276 { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }

```

(End of definition for `\__fp_neg_sign:N`.)

`\__fp_kind:w` Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for `nan`, 4 for tuples.

```

24277 \cs_new:Npn \__fp_kind:w #1
24278 {
24279   \__fp_if_type_fp:NTwFw
24280   #1 \__fp_use_ii_until_s:nnw
24281   \s__fp { \__fp_use_i_until_s:nw 4 }
24282   \s__fp_stop
24283 }

```

(End of definition for `\__fp_kind:w`.)

73.5 Overflow, underflow, and exact zero

<pre> __fp_sanitizer:Nw __fp_sanitizer:wN __fp_sanitizer_zero:w </pre>	Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm \infty$. The functions <code>__fp_underflow:w</code> and <code>__fp_overflow:w</code> are defined in <code>l3fp-traps</code>.

```

24284 \cs_new:Npn \__fp_sanitizew #1 #2\__fp_sep:
24285 {
24286   \if_case:w
24287     \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
24288     \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
24289     \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
24290   \or: \exp_after:wN \__fp_overflow:w
24291   \or: \exp_after:wN \__fp_underflow:w
24292   \or: \exp_after:wN \__fp_sanitizew_zero:w
24293   \fi:
24294   \s__fp \__fp_chk:w 1 #1 {#2}
24295 }
24296 \cs_new:Npn \__fp_sanitizewN #1\__fp_sep: #2 { \__fp_sanitizewN #2 #1\__fp_sep: }
24297 \cs_new:Npn \__fp_sanitizew_zero:w \s__fp \__fp_chk:w #1 #2 #3\__fp_sep:
24298 { \c_zero_fp }

```

(End of definition for _fp_sanitize:Nw, _fp_sanitize:wN, and _fp_sanitize zero:w.)

73.6 Expanding after a floating point number

```

\__fp_exp_after_o:w
\__fp_exp_after_f:nw

```

`\_fp_exp_after_o:w` *<floating point>*
`\_fp_exp_after_f:nw` *{(tokens)}* *<floating point>*
 Places *<tokens>* (empty in the case of `\_fp_exp_after_o:w`) between the *<floating point>* and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

24299 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
24300 {
24301   \if_meaning:w 1 #1
24302     \exp_after:wN \__fp_exp_after_normal:nNNw
24303   \else:
24304     \exp_after:wN \__fp_exp_after_special:nNNw
24305   \fi:
24306   { }
24307   #1
24308 }
24309 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
24310 {
24311   \if_meaning:w 1 #2
24312     \exp_after:wN \__fp_exp_after_normal:nNNw
24313   \else:
24314     \exp_after:wN \__fp_exp_after_special:nNNw
24315   \fi:
24316   { \exp:w \exp_end_continue_f:w #1 }
24317   #2

```

24318 }

(End of definition for `\__fp_exp_after_o:w` and `\__fp_exp_after_f:nw`.)

`\__fp_exp_after_special:nNw` `\__fp_exp_after_special:nNw` {`<after>`} `<case>` `<sign>` `<scan mark>` `\__fp_sep:`
Special floating point numbers are easy to jump over since they contain few tokens.

```
24319 \cs_new:Npn \__fp_exp_after_special:nNw #1#2#3#4\__fp_sep:
24320 {
24321   \exp_after:wN \s__fp
24322   \exp_after:wN \__fp_chk:w
24323   \exp_after:wN #2
24324   \exp_after:wN #3
24325   \exp_after:wN #4
24326   \exp_after:wN \__fp_sep:
24327   #1
24328 }
```

(End of definition for `\__fp_exp_after_special:nNw`.)

`\__fp_exp_after_normal:nNw` For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by `,`). That may be changed some day.

```
24329 \cs_new:Npn \__fp_exp_after_normal:nNw #1 1 #2 #3 #4#5#6#7\__fp_sep:
24330 {
24331   \exp_after:wN \__fp_exp_after_normal:Nwwwww
24332   \exp_after:wN #2
24333   \int_value:w #3   \exp_after:wN \__fp_sep:
24334   \int_value:w 1 #4 \exp_after:wN \__fp_sep:
24335   \int_value:w 1 #5 \exp_after:wN \__fp_sep:
24336   \int_value:w 1 #6 \exp_after:wN \__fp_sep:
24337   \int_value:w 1 #7 \exp_after:wN \__fp_sep: #1
24338 }
24339 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
24340   #1 #2\__fp_sep: 1 #3 \__fp_sep: 1 #4 \__fp_sep: 1 #5 \__fp_sep: 1 #6 \__fp_sep:
24341   { \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} \__fp_sep: }
```

(End of definition for `\__fp_exp_after_normal:nNw`.)

73.7 Other floating point types

`\s__fp_tuple` Floating point tuples take the form `\s__fp_tuple \__fp_tuple_chk:w { <fp 1> <fp 2>`
`\__fp_tuple_chk:w` `...} \__fp_sep:` where each `<fp>` is a floating point number or tuple, hence ends with
`\c__fp_empty_tuple_fp` `\__fp_sep:` itself. When a tuple is typeset, `\__fp_tuple_chk:w` produces an error, just
like usual floating point numbers. Tuples may have zero or one element.

```
24342 \scan_new:N \s__fp_tuple
24343 \cs_new_protected:Npn \__fp_tuple_chk:w #1 \__fp_sep:
24344   { \__fp_misused:n { \s__fp_tuple \__fp_tuple_chk:w #1 \__fp_sep: } }
24345 \tl_const:Nn \c__fp_empty_tuple_fp
24346   { \s__fp_tuple \__fp_tuple_chk:w { } \__fp_sep: }
```

(End of definition for `\s__fp_tuple`, `\__fp_tuple_chk:w`, and `\c__fp_empty_tuple_fp`.)

`\__fp_tuple_count:w` Count the number of items in a tuple of floating points by counting `\__fp_sep:s`. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

24347 \cs_new:Npn \__fp_array_count:n #1
24348 { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} \__fp_sep: }
24349 \cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 \__fp_sep:
24350 {
24351   \int_value:w \__fp_int_eval:w 0
24352   \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } \__fp_sep:
24353   \prg_break_point:
24354   \__fp_int_eval_end:
24355 }
24356 \cs_new:Npn \__fp_tuple_count_loop:Nw #1#2\__fp_sep:
24357 { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }

```

(End of definition for `\__fp_tuple_count:w`, `\__fp_array_count:n`, and `\__fp_tuple_count_loop:Nw`.)

`\__fp_if_type_fp:NTwFw` Used as `\__fp_if_type_fp:NTwFw <marker> {<true code>} \s__fp {<false code>} \s__fp_stop`, this test whether the `<marker>` is `\s__fp` or not and runs the appropriate `<code>`. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```

24358 \cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \s__fp_stop {#2}

```

(End of definition for `\__fp_if_type_fp:NTwFw`.)

`\__fp_array_if_all_fp:nTF` True if all items are floating point numbers. Used for min.
`\__fp_array_if_all_fp_loop:w`

```

24359 \cs_new:Npn \__fp_array_if_all_fp:nTF #1
24360 {
24361   \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } \__fp_sep:
24362   \prg_break_point: \use_i:nn
24363 }
24364 \cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 \__fp_sep:
24365 {
24366   \__fp_if_type_fp:NTwFw
24367   #1 \__fp_array_if_all_fp_loop:w
24368   \s__fp { \prg_break:n \use_iii:nnn }
24369   \s__fp_stop
24370 }

```

(End of definition for `\__fp_array_if_all_fp:nTF` and `\__fp_array_if_all_fp_loop:w`.)

`\__fp_type_from_scan:N` Used as `\__fp_type_from_scan:N <token>`. Grabs the pieces of the stringified `<token>` which lies after the first `s__fp`. If the `<token>` does not contain that string, the result is `_?`.
`\__fp_type_from_scan_other:N`
`\__fp_type_from_scan:w`

```

24371 \cs_new:Npn \__fp_type_from_scan:N #1
24372 {
24373   \__fp_if_type_fp:NTwFw
24374   #1 { }
24375   \s__fp { \__fp_type_from_scan_other:N #1 }
24376   \s__fp_stop
24377 }
24378 \cs_new:Npe \__fp_type_from_scan_other:N #1
24379 {
24380   \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w

```

```

24381 \exp_not:N \token_to_str:N #1 \s__fp_mark
24382 \tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
24383 }
24384 \exp_last_unbraced:NNNNo
24385 \cs_new:Npn \__fp_type_from_scan:w #1
24386 { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}

```

(End of definition for __fp_type_from_scan:N, __fp_type_from_scan_other:N, and __fp_type_from_scan:w.)

```

\__fp_change_func_type:NNN
\__fp_change_func_type_aux:w
\__fp_change_func_type_chk:NNN

```

Arguments are $\langle type\ marker \rangle$ $\langle function \rangle$ $\langle recovery \rangle$. This gives the function obtained by placing the type after @@. If the function is not defined then $\langle recovery \rangle$ $\langle function \rangle$ is used instead; however that test is not run when the $\langle type\ marker \rangle$ is $\backslash s\_fp$.

```

24387 \cs_new:Npn \__fp_change_func_type:NNN #1#2#3
24388 {
24389   \__fp_if_type_fp:NTwFw
24390   #1 #2
24391   \s__fp
24392   {
24393     \exp_after:wN \__fp_change_func_type_chk:NNN
24394     \cs:w
24395     __fp \__fp_type_from_scan_other:N #1
24396     \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
24397     \cs_end:
24398     #2 #3
24399   }
24400   \s__fp_stop
24401 }
24402 \exp_last_unbraced:NNNNo
24403 \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
24404 \cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
24405 {
24406   \if_meaning:w \scan_stop: #1
24407   \exp_after:wN #3 \exp_after:wN #2
24408   \else:
24409   \exp_after:wN #1
24410   \fi:
24411 }

```

(End of definition for __fp_change_func_type:NNN, __fp_change_func_type_aux:w, and __fp_change_func_type_chk:NNN.)

```

\__fp_exp_after_any_f:Nnw
\__fp_exp_after_any_f:nw
\__fp_exp_after_expr_stop_f:nw

```

The Nnw function simply dispatches to the appropriate $\backslash \_fp\_exp\_after \dots f:nw$ with “...” (either empty or $\_ \langle type \rangle$) extracted from #1, which should start with $\backslash s\_fp$. If it doesn’t start with $\backslash s\_fp$ the function $\backslash \_fp\_exp\_after\_?_f:nw$ defined in `l3fp-parse` gives an error; another special $\langle type \rangle$ is `stop`, useful for loops, see below. The nw function has an important optimization for floating points numbers; it also fetches its type marker #2 from the floating point.

```

24412 \cs_new:Npn \__fp_exp_after_any_f:Nnw #1
24413 { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
24414 \cs_new:Npn \__fp_exp_after_any_f:nw #1#2
24415 {
24416   \__fp_if_type_fp:NTwFw
24417   #2 \__fp_exp_after_f:nw

```



```

24418 \s__fp { \__fp_exp_after_any_f:Nnw #2 }
24419 \s__fp_stop
24420 {#1} #2
24421 }
24422 \cs_new_eq:NN \__fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End of definition for `\__fp_exp_after_any_f:Nnw`, `\__fp_exp_after_any_f:nw`, and `\__fp_exp_after_expr_stop_f:nw`.)

`\__fp_exp_after_tuple_o:w` The loop works by using the `n` argument of `\__fp_exp_after_any_f:nw` to place the
`\__fp_exp_after_tuple_f:nw` loop macro after the next item in the tuple and expand it.
`\__fp_exp_after_array_f:w`

```

\__fp_exp_after_array_f:w
⟨fp1⟩ \__fp_sep:
...
⟨fpn⟩ \__fp_sep:
\s__fp_expr_stop

24423 \cs_new:Npn \__fp_exp_after_tuple_o:w
24424 { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
24425 \cs_new:Npn \__fp_exp_after_tuple_f:nw
24426 #1 \s__fp_tuple \__fp_tuple_chk:w #2 \__fp_sep:
24427 {
24428 \exp_after:wN \s__fp_tuple
24429 \exp_after:wN \__fp_tuple_chk:w
24430 \exp_after:wN {
24431 \exp:w \exp_end_continue_f:w
24432 \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
24433 \exp_after:wN }
24434 \exp_after:wN \__fp_sep:
24435 \exp:w \exp_end_continue_f:w #1
24436 }
24437 \cs_new:Npn \__fp_exp_after_array_f:w
24438 { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }

```

(End of definition for `\__fp_exp_after_tuple_o:w`, `\__fp_exp_after_tuple_f:nw`, and `\__fp_exp_after_array_f:w`.)

73.8 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \__fp_pack:NNNNnw #1 #2#3#4#5 #6 \__fp_sep: { {#2#3#4#5} {#6} }
\exp_after:wN \__fp_pack:NNNNnw
\__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 \__fp_sep:

```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by $\text{T}_{\text{E}}\text{X}$'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 \__fp_sep: {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute 12345×66778899 . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
\exp_after:wN \__fp_pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \__fp_pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 5 0000 0000
+ 12345 * 8899 \__fp_sep:

```

The `\exp_after:wN` triggers `\int_value:w \__fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w \__fp_int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the `\__fp_sep:` delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ {5 digits}` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure `TeX` floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

```

\__fp_pack:NNNNNw
\c__fp_trailing_shift_int
\c__fp_middle_shift_int
\c__fp_leading_shift_int

```

This set of shifts allows for computations involving results in the range $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$. Shifted values all have exactly 9 digits.

```

24439 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
24440 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
24441 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
24442 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6\__fp_sep:
24443 { + #1#2#3#4#5 \__fp_sep: {#6} }

```

(End of definition for `\__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw
\c__fp_big_trailing_shift_int
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int

```

This set of shifts allows for computations involving results in the range $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to `TeX`’s limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in `TeX`.

```

24444 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
24445 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
24446 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
24447 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7\__fp_sep:
24448 { + #1#2#3#4#5#6 \__fp_sep: {#7} }

```

(End of definition for `\__fp_pack_big:NNNNNNw` and others.)

`\__fp_pack_Bigg:NNNNNNw`
`\c_fp_Bigg_trailing_shift_int`
`\c__fp_Bigg_middle_shift_int`
`\c_fp_Bigg_leading_shift_int`

This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```

24449 \int_const:Nn \c__fp_Bigg_leading_shift_int { - 20 0000 }
24450 \int_const:Nn \c__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
24451 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
24452 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7\__fp_sep:
24453 { + #1#2#3#4#5#6 \__fp_sep: {#7} }

```

(End of definition for `\__fp_pack_Bigg:NNNNNNw` and others.)

`\__fp_pack_twice_four:wNNNNNNNN`

`\__fp_pack_twice_four:wNNNNNNNN` *<tokens>* `\__fp_sep: <≥ 8 digits>`
Grabs two sets of 4 digits and places them before the `\__fp_sep:` delimiter. Putting several copies of this function before a `\__fp_sep:` packs more digits since each takes the digits packed by the others in its first argument.

```

24454 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1\__fp_sep: #2#3#4#5 #6#7#8#9
24455 { #1 {#2#3#4#5} {#6#7#8#9} \__fp_sep: }

```

(End of definition for `\__fp_pack_twice_four:wNNNNNNNN`.)

`\__fp_pack_eight:wNNNNNNNN`

`\__fp_pack_eight:wNNNNNNNN` *<tokens>* `\__fp_sep: <≥ 8 digits>`
Grabs one set of 8 digits and places them before the `\__fp_sep:` delimiter as a single group. Putting several copies of this function before a `\__fp_sep:` packs more digits since each takes the digits packed by the others in its first argument.

```

24456 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1\__fp_sep: #2#3#4#5 #6#7#8#9
24457 { #1 {#2#3#4#5#6#7#8#9} \__fp_sep: }

```

(End of definition for `\__fp_pack_eight:wNNNNNNNN`.)

`\__fp_basics_pack_low:NNNNNw`
`\__fp_basics_pack_high:NNNNNw`
`\__fp_basics_pack_high_carry:w`

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `\__fp_basics_pack_high_carry:w` is always followed by four times {0000}.

This is used in l3fp-basics and l3fp-extended.

```

24458 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6\__fp_sep:
24459 { + #1 - 1 \__fp_sep: {#2#3#4#5} {#6} \__fp_sep: }
24460 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6\__fp_sep:
24461 {
24462   \if_meaning:w 2 #1
24463     \__fp_basics_pack_high_carry:w
24464   \fi:
24465   \__fp_sep: {#2#3#4#5} {#6}
24466 }
24467 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: \__fp_sep: #1
24468 { \fi: + 1 \__fp_sep: {1000} }

```

(End of definition for `\__fp_basics_pack_low:NNNNNw`, `\__fp_basics_pack_high:NNNNNw`, and `\__fp_basics_pack_high_carry:w`.)

_fp_basics_pack_weird_low:NNNNw
 _fp_basics_pack_weird_high:NNNNNNNNw

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```

24469 \cs_new:Npn \_fp\_basics\_pack\_weird\_low:NNNNw #1 #2#3#4 #5\_fp\_sep:
24470 {
24471   \if_meaning:w 2 #1
24472     + 1
24473   \fi:
24474   \_fp\_int\_eval\_end:
24475   #2#3#4\_fp\_sep: {#5} \_fp\_sep:
24476 }
24477 \cs_new:Npn \_fp\_basics\_pack\_weird\_high:NNNNNNNNw
24478   1 #1#2#3#4 #5#6#7#8 #9\_fp\_sep: { \_fp\_sep: {#1#2#3#4} {#5#6#7#8} {#9} }
```

(End of definition for _fp_basics_pack_weird_low:NNNNw and _fp_basics_pack_weird_high:NNNNNNNNw.)

73.9 Decimate (dividing by a power of 10)

_fp_decimate:nNnnnn

_fp_decimate:nNnnnn {<shift>} {<f₁>
 {<X₁>} {<X₂>} {<X₃>} {<X₄>}

Each <X_i> consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. <f₁> is called as follows:

<f₁> <rounding> {<X'₁>} {<X'₂>} <extra-digits> _fp_sep:

where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle \text{shift} \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle \text{extra-digits} \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The <rounding> digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, <rounding> is 1 (not 0), and <X'₁> and <X'₂> are both zero.

If the shift is 1, the <rounding> digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the <rounding> digit to be placed after the <X'_i>, but the choice we make involves less reshuffling.

Note that this function treats negative <shift> as 0.

```

24479 \cs_new:Npn \_fp\_decimate:nNnnnn #1
24480 {
24481   \cs:w
24482     \_fp\_decimate_
24483     \if_int_compare:w \_fp\_int\_eval:w #1 > \c\_fp\_prec\_int
24484       tiny
24485     \else:
24486       \_fp\_int\_to\_roman:w \_fp\_int\_eval:w #1
24487     \fi:
24488     :Nnnnn
24489   \cs_end:
24490 }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End of definition for `\_fp\_decimate:Nnnnn`.)

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
\_fp\_decimate_:Nnnnn
\_fp\_decimate\_tiny:Nnnnn
24491 \cs_new:Npn \_fp\_decimate_:Nnnnn #1 #2#3#4#5
24492 { #1 0 {#2#3} {#4#5} \_fp\_sep: }
24493 \cs_new:Npn \_fp\_decimate\_tiny:Nnnnn #1 #2#3#4#5
24494 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 \_fp\_sep: }
```

(End of definition for `\_fp\_decimate_:Nnnnn` and `\_fp\_decimate\_tiny:Nnnnn`.)

```
\_fp\_decimate\_auxi:Nnnnn      \_fp\_decimate\_auxi:Nnnnn  $\langle f_1 \rangle$  { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ }
\_fp\_decimate\_auxii:Nnnnn      Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into
\_fp\_decimate\_auxiii:Nnnnn      two blocks of 8. The sixteen functions are very similar, and defined through \_fp\_
\_fp\_decimate\_auxiv:Nnnnn      tmp:w. The arguments are as follows: #1 indicates which function is being defined;
\_fp\_decimate\_auxv:Nnnnn      after one step of expansion, #2 yields the “extra digits” which are then converted by
\_fp\_decimate\_auxvi:Nnnnn      \_fp\_round\_digit:Nw to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits
\_fp\_decimate\_auxvii:Nnnnn      to avoid overflowing TeX’s integers). This triggers the f-expansion of \_fp\_decimate\_
\_fp\_decimate\_auxviii:Nnnnn      pack:nnnnnnnnnw,11 responsible for building two blocks of 8 digits, and removing the
\_fp\_decimate\_auxix:Nnnnn      rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in
\_fp\_decimate\_auxxx:Nnnnn      such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.
\_fp\_decimate\_auxxi:Nnnnn
\_fp\_decimate\_auxxii:Nnnnn
\_fp\_decimate\_auxxiii:Nnnnn
\_fp\_decimate\_auxxiv:Nnnnn
\_fp\_decimate\_auxxv:Nnnnn
\_fp\_decimate\_auxxvi:Nnnnn
24495 \cs_new:Npn \_fp\_tmp:w #1 #2 #3
24496 {
24497   \cs_new:cpn { \_fp\_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
24498   {
24499     \exp_after:wN ##1
24500     \int_value:w
24501     \exp_after:wN \_fp\_round\_digit:Nw #2 \_fp\_sep:
24502     \_fp\_decimate\_pack:nnnnnnnnnw #3 \_fp\_sep:
24503   }
24504 }
24505 \_fp\_tmp:w {i}   {\use\_none:nnn   #50}{ 0{#2}#3{#4}#5      }
24506 \_fp\_tmp:w {ii}  {\use\_none:nn    #5 }{ 00{#2}#3{#4}#5      }
24507 \_fp\_tmp:w {iii} {\use\_none:n     #5 }{ 000{#2}#3{#4}#5      }
24508 \_fp\_tmp:w {iv}  {                #5 }{ {0000}#2{#3}#4 #5    }
24509 \_fp\_tmp:w {v}   {\use\_none:nnn   #4#5 }{ 0{0000}#2{#3}#4 #5    }
24510 \_fp\_tmp:w {vi}  {\use\_none:nn    #4#5 }{ 00{0000}#2{#3}#4 #5    }
24511 \_fp\_tmp:w {vii} {\use\_none:n     #4#5 }{ 000{0000}#2{#3}#4 #5    }
24512 \_fp\_tmp:w {viii}{                #4#5 }{ {0000}0000{#2}#3 #4 #5    }
24513 \_fp\_tmp:w {ix}  {\use\_none:nnn   #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5    }
24514 \_fp\_tmp:w {x}   {\use\_none:nn    #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5    }
24515 \_fp\_tmp:w {xi}  {\use\_none:n     #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5    }
24516 \_fp\_tmp:w {xii} {                #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5    }
24517 \_fp\_tmp:w {xiii}{\use\_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5    }
24518 \_fp\_tmp:w {xiv} {\use\_none:nn    #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5    }
24519 \_fp\_tmp:w {xv}  {\use\_none:n     #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5    }
24520 \_fp\_tmp:w {xvi} {                #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5 }
```

(End of definition for `\_fp\_decimate\_auxi:Nnnnn` and others.)

¹¹No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

`\__fp_decimate_pack:nnnnnnnnnw`

The computation of the *<rounding>* digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a `\__fp_sep:`.

```
24521 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
24522 { \__fp_decimate_pack:nnnnnw { #1#2#3#4#5 } }
24523 \cs_new:Npn \__fp_decimate_pack:nnnnnw #1 #2#3#4#5#6
24524 { {#1} {#2#3#4#5#6} }
```

(End of definition for `\__fp_decimate_pack:nnnnnnnnnw`.)

73.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in `l3fp` must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if_case:w <integer> \exp_stop_f:
  \__fp_case_return_o:Nw <fp var>
\or: \__fp_case_use:nw {<some computation>}
\or: \__fp_case_return_same_o:w
\or: \__fp_case_return:nw {<something>}
\fi:
<junk>
<floating point>
```

In this example, the case 0 returns the floating point *<fp var>*, expanding once after that floating point. Case 1 does *<some computation>* using the *<floating point>* (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the *<floating point>* without modifying it, removing the *<junk>* and expanding once after. Case 3 closes the conditional, removes the *<junk>* and the *<floating point>*, and expands *<something>* next. In other cases, the “*<junk>*” is expanded, performing some other operation on the *<floating point>*. We provide similar functions with two trailing *<floating points>*.

`\__fp_case_use:nw`

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
24525 \cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
```

(End of definition for `\__fp_case_use:nw`.)

`\__fp_case_return:nw` This function ends a $\TeX$ conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a `\__fp_sep:`. This, in turn, means that the `\junk` may not contain `\__fp_sep:s`.

```
24526 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 \__fp_sep: { \fi: #1 }
```

(End of definition for __fp_case_return:nw.)

`\__fp_case_return_o:Nw` This function ends a $\TeX$ conditional, removes junk and a floating point, and returns its first argument (an `\fp var`) then expands once after it.

```
24527 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 \__fp_sep:
24528 { \fi: \exp_after:wN #1 }
```

(End of definition for __fp_case_return_o:Nw.)

`\__fp_case_return_same_o:w` This function ends a $\TeX$ conditional, removes junk, and returns the following floating point, expanding once after it.

```
24529 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
24530 { \fi: \__fp_exp_after_o:w \s__fp }
```

(End of definition for __fp_case_return_same_o:w.)

`\__fp_case_return_o:Nww` Same as `\__fp_case_return_o:Nw` but with two trailing floating points.

```
24531 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 \__fp_sep: #5 \__fp_sep:
24532 { \fi: \exp_after:wN #1 }
```

(End of definition for __fp_case_return_o:Nww.)

`\__fp_case_return_i_o:ww` Similar to `\__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
24533 \cs_new:Npn \__fp_case_return_i_o:ww
24534 #1 \fi: #2 \s__fp #3 \__fp_sep: \s__fp #4 \__fp_sep:
24535 { \fi: \__fp_exp_after_o:w \s__fp #3 \__fp_sep: }
24536 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 \__fp_sep:
24537 { \fi: \__fp_exp_after_o:w }
```

(End of definition for __fp_case_return_i_o:ww and __fp_case_return_ii_o:ww.)

73.11 Integer floating points

`\__fp_int_p:w` Tests if the floating point argument is an integer. For normal floating point numbers, `\__fp_int:wTF` this holds if the rounding digit resulting from `\__fp_decimate:nNnnnn` is 0.

```
24538 \prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4\__fp_sep:
24539 { TF , T , F , p }
24540 {
24541   \if_case:w #1 \exp_stop_f:
24542     \prg_return_true:
24543   \or:
24544     \if_charcode:w 0
24545       \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
24546       \__fp_use_i_until_s:nw #4
24547       \prg_return_true:
24548     \else:
```

```

24549         \prg_return_false:
24550         \fi:
24551     \else: \prg_return_false:
24552     \fi:
24553 }

```

(End of definition for `\__fp_int:wTF`.)

73.12 Small integer floating points

`\__fp_small_int:wTF` Tests if the floating point argument is an integer or $\pm\infty$. If so, it is clipped to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the `<true code>`. Otherwise, the `<false code>` is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the `<false code>`. If it is, then the integer is `#2 #3`; use `#3` if `#2` vanishes and otherwise `108`.

```

24554 \cs_new:Npn \__fp_small_int:wTF \s_fp \__fp_chk:w #1#2
24555 {
24556     \if_case:w #1 \exp_stop_f:
24557         \__fp_case_return:nw { \__fp_small_int_true:wTF 0 \__fp_sep: }
24558     \or: \exp_after:wN \__fp_small_int_normal:NnwTF
24559     \or:
24560         \__fp_case_return:nw
24561         {
24562             \exp_after:wN \__fp_small_int_true:wTF \int_value:w
24563             \if_meaning:w 2 #2 - \fi: 1 0000 0000 \__fp_sep:
24564         }
24565     \else: \__fp_case_return:nw \use_ii:nn
24566     \fi:
24567     #2
24568 }
24569 \cs_new:Npn \__fp_small_int_true:wTF #1\__fp_sep: #2#3 { #2 {#1} }
24570 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3\__fp_sep:
24571 {
24572     \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
24573     \__fp_small_int_test:NnnwNw
24574     #3 #1
24575 }
24576 \cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4\__fp_sep: #5
24577 {
24578     \if_meaning:w 0 #1
24579     \exp_after:wN \__fp_small_int_true:wTF
24580     \int_value:w \if_meaning:w 2 #5 - \fi:
24581     \if_int_compare:w #2 > \c_zero_int
24582         1 0000 0000
24583     \else:
24584         #3
24585     \fi:
24586     \exp_after:wN \__fp_sep:
24587 \else:
24588     \exp_after:wN \use_ii:nn
24589 \fi:
24590 }

```


(End of definition for `\__fp_small_int:wTF` and others.)

73.13 Fast string comparison

`\__fp_str_if_eq:nn` A private version of the low-level string comparison function.

```
24591 \cs_new_eq:NN \__fp_str_if_eq:nn \tex_strcmp:D
```

(End of definition for `\__fp_str_if_eq:nn`.)

73.14 Name of a function from its l3fp-parse name

`\__fp_func_to_name:N` The goal is to convert for instance `\__fp_sin_o:w` to `sin`. This is used in error messages
`\__fp_func_to_name_aux:w` hence does not need to be fast.

```
24592 \cs_new:Npn \__fp_func_to_name:N #1
24593 {
24594   \exp_last_unbraced:Nf
24595   \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
24596 }
24597 \cs_set_protected:Npn \__fp_tmp:w #1 #2
24598 { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
24599 \exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
24600 { \tl_to_str:n { _o: } }
```

(End of definition for `\__fp_func_to_name:N` and `\__fp_func_to_name_aux:w`.)

73.15 Messages

Using a floating point directly is an error.

```
24601 \msg_new:nnnn { fp } { misused }
24602 { A~floating~point~with~value~'~#1'~was~misused. }
24603 {
24604   To~obtain~the~value~of~a~floating~point~variable,~use~
24605   '\token_to_str:N \fp_to_decimal:N',~
24606   '\token_to_str:N \fp_to_tl:N',~or~other~
24607   conversion~functions.
24608 }
24609 \prop_gput:Nnn \g_msg_module_name_prop { fp } { LaTeX }
24610 \prop_gput:Nnn \g_msg_module_type_prop { fp } { }
24611 \code
```

Chapter 74

l3fp-traps implementation

24612 $\langle *code \rangle$

24613 $\langle @@=fp \rangle$

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

74.1 Flags

Flags to denote exceptions.

```
\l_fp_invalid_operation_flag 24614 \flag_new:N \l_fp_invalid_operation_flag
\l_fp_division_by_zero_flag 24615 \flag_new:N \l_fp_division_by_zero_flag
\l_fp_overflow_flag          24616 \flag_new:N \l_fp_overflow_flag
\l_fp_underflow_flag         24617 \flag_new:N \l_fp_underflow_flag
```

(End of definition for `\l_fp_invalid_operation_flag` and others. These variables are documented on page 277.)

74.2 Traps

Exceptions can be trapped to obtain custom behavior. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an `N`-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the `inexact` exception is entirely ignored.

The behavior when an exception occurs is controlled by the definitions of the functions

- `\__fp_invalid_operation:nw`,

- __fp_invalid_operation_o:Nww,
- __fp_invalid_operation_tl_o:ff,
- __fp_division_by_zero_o:Nnw,
- __fp_division_by_zero_o:NNww,
- __fp_overflow:w,
- __fp_underflow:w.

Rather than changing them directly, we provide a user interface as \fp_trap:nn {*exception*} {*way of trapping*}, where the *way of trapping* is one of error, flag, or none.

We also provide __fp_invalid_operation_o:nw, defined in terms of __fp_invalid_operation:nnw.

\fp_trap:nn

```

24618 \cs_new_protected:Npn \fp_trap:nn #1#2
24619 {
24620   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
24621   {
24622     \clist_if_in:nnTF
24623     { invalid_operation , division_by_zero , overflow , underflow }
24624     {#1}
24625     {
24626       \msg_error:nnee { fp }
24627       { unknown-fpu-trap-type } {#1} {#2}
24628     }
24629     {
24630       \msg_error:nne
24631       { fp } { unknown-fpu-exception } {#1}
24632     }
24633   }
24634 }
```

(End of definition for \fp_trap:nn. This function is documented on page 277.)

_fp_trap_invalid_operation_set_error: We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```

\_fp_trap_invalid_operation_set_error:
\_fp_trap_invalid_operation_set_flag:
\_fp_trap_invalid_operation_set_none:
\_fp_trap_invalid_operation_set:N
24635 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
24636 { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
24637 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
24638 { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
24639 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
24640 { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
24641 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
24642 {
24643   \exp_args:Nno \use:n
24644   { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3\_fp_sep: }
24645   {
24646     #1
24647     \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3\_fp_sep: } } { }
```

```

24648     \flag_ensure_raised:N \l_fp_invalid_operation_flag
24649     ##1
24650   }
24651 \exp_args:Nno \use:n
24652 { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2\__fp_sep: ##3\__fp_sep: }
24653 {
24654   #1
24655   \__fp_error:nffn { invalid-ii }
24656   { \fp_to_tl:n { ##2\__fp_sep: } }
24657   { \fp_to_tl:n { ##3\__fp_sep: } }
24658   {##1}
24659   \flag_ensure_raised:N \l_fp_invalid_operation_flag
24660   \exp_after:wN \c_nan_fp
24661 }
24662 \exp_args:Nno \use:n
24663 { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
24664 {
24665   #1
24666   \__fp_error:nffn { invalid } {##1} {##2} { }
24667   \flag_ensure_raised:N \l_fp_invalid_operation_flag
24668   \exp_after:wN \c_nan_fp
24669 }
24670 }

```

(End of definition for __fp_trap_invalid_operation_set_error: and others.)

__fp_trap_division_by_zero_set_error: We provide three types of trapping for invalid operations and division by zero: either
 __fp_trap_division_by_zero_set_flag: produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
 __fp_trap_division_by_zero_set_none: flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
 __fp_trap_division_by_zero_set:N nan.

```

24671 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
24672 { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
24673 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
24674 { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
24675 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
24676 { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
24677 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
24678 {
24679   \exp_args:Nno \use:n
24680   { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3\__fp_sep: }
24681   {
24682     #1
24683     \__fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3\__fp_sep: } } { }
24684     \flag_ensure_raised:N \l_fp_division_by_zero_flag
24685     \exp_after:wN ##1
24686   }
24687 \exp_args:Nno \use:n
24688 {
24689   \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3\__fp_sep: ##4\__fp_sep:
24690 }
24691 {
24692   #1
24693   \__fp_error:nffn { zero-div-ii }
24694   { \fp_to_tl:n { ##3\__fp_sep: } }

```

```

24695         { \fp_to_tl:n { ##4\__fp_sep: } }
24696         {##2}
24697         \flag_ensure_raised:N \l_fp_division_by_zero_flag
24698         \exp_after:wN ##1
24699     }
24700 }

```

(End of definition for `\__fp_trap_division_by_zero_set_error:` and others.)

`\__fp_trap_overflow_set_error:` Just as for invalid operations and division by zero, the three different behaviors are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\__fp_overflow:w` and `\__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for T_EX, and `\__fp_overflow:w` receives $\pm\infty$ (`\__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

24701 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
24702 { \__fp_trap_overflow_set:N \prg_do_nothing: }
24703 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
24704 { \__fp_trap_overflow_set:N \use_none:nnnnn }
24705 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
24706 { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
24707 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
24708 { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
24709 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
24710 { \__fp_trap_underflow_set:N \prg_do_nothing: }
24711 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
24712 { \__fp_trap_underflow_set:N \use_none:nnnnn }
24713 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
24714 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
24715 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
24716 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
24717 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
24718 {
24719     \exp_args:Nno \use:n
24720     { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3\__fp_sep: }
24721     {
24722         #1
24723         \__fp_error:nffn
24724         { flow \if_meaning:w 1 ##1 -to \fi: }
24725         { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3\__fp_sep: } }
24726         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
24727         {#2}
24728         \flag_ensure_raised:c { l_fp_#2_flag }
24729         #3 ##2
24730     }
24731 }

```

(End of definition for `\__fp_trap_overflow_set_error:` and others.)

`\__fp_invalid_operation:nnw` Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on

```

\__fp_invalid_operation_o:Nnw
\__fp_invalid_operation_tl_o:ff
\__fp_division_by_zero_o:Nnw
\__fp_division_by_zero_o:NNnw
\__fp_overflow:w
\__fp_underflow:w

```

their flag.

```

24732 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3\__fp_sep: { }
24733 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2\__fp_sep: #3\__fp_sep: { }
24734 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
24735 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3\__fp_sep: { }
24736 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3\__fp_sep: #4\__fp_sep: { }
24737 \cs_new:Npn \__fp_overflow:w { }
24738 \cs_new:Npn \__fp_underflow:w { }
24739 \fp_trap:nn { invalid_operation } { error }
24740 \fp_trap:nn { division_by_zero } { flag }
24741 \fp_trap:nn { overflow } { flag }
24742 \fp_trap:nn { underflow } { flag }

```

(End of definition for `\__fp_invalid_operation:nnw` and others.)

`\__fp_invalid_operation_o:nw` Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and
`\__fp_invalid_operation_o:fw` expanding after.

```

24743 \cs_new:Npn \__fp_invalid_operation_o:nw
24744   { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
24745 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End of definition for `\__fp_invalid_operation_o:nw`.)

74.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn 24746 \cs_new:Npn \__fp_error:nnnn
\__fp_error:nffn 24747   { \msg_expandable_error:nnnnn { fp } }
\__fp_error:nfff 24748 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

```

(End of definition for `\__fp_error:nnnn`.)

```

\__fp_error_num_args:nnnn
\__fp_error_num_args:ffff
24749 \cs_new:Npn \__fp_error_num_args:nnnn #1#2#3#4
24750   {
24751     \int_compare:nNnTF {#2} = {#3}
24752       { \msg_expandable_error:nnnnn { fp } { num-args-eq } {#1} {#2} {#4} }
24753       { \msg_expandable_error:nnnnn { fp } { num-args } {#1} {#2} {#3} {#4} }
24754   }
24755 \cs_generate_variant:Nn \__fp_error_num_args:nnnn { ffff }
24756 \msg_new:nnn { fp } { num-args-eq }
24757   { #1()~needs~#2~arguments,~got~#3. }
24758 \msg_new:nnn { fp } { num-args }
24759   { #1()~needs~#2~to~#3~arguments,~got~#4. }

```

(End of definition for `\__fp_error_num_args:nnnn`.)

74.4 Messages

Some messages.

```
24760 \msg_new:nnnn { fp } { unknown-fpu-exception }
24761 {
24762     The-FPU-exception-~'~#1'~is-not-known:~
24763     that~trap~will~never~be~triggered.
24764 }
24765 {
24766     The-only-exceptions-to-which-traps-can-be-attached-are \\
24767     \iow_indent:n
24768     {
24769         * ~ invalid_operation \\
24770         * ~ division_by_zero \\
24771         * ~ overflow \\
24772         * ~ underflow
24773     }
24774 }
24775 \msg_new:nnnn { fp } { unknown-fpu-trap-type }
24776 { The-FPU-trap-type-~'~#2'~is-not-known. }
24777 {
24778     The-trap-type-must-be-one-of \\
24779     \iow_indent:n
24780     {
24781         * ~ error \\
24782         * ~ flag \\
24783         * ~ none
24784     }
24785 }
24786 \msg_new:nnn { fp } { flow }
24787 { An ~ #3 ~ occurred. }
24788 \msg_new:nnn { fp } { flow-to }
24789 { #1 ~ #3 ed ~ to ~ #2 . }
24790 \msg_new:nnn { fp } { zero-div }
24791 { Division-by-zero-in~ #1 (#2) }
24792 \msg_new:nnn { fp } { zero-div-ii }
24793 { Division-by-zero-in~ (#1) #3 (#2) }
24794 \msg_new:nnn { fp } { invalid }
24795 { Invalid-operation~ #1 (#2) }
24796 \msg_new:nnn { fp } { invalid-ii }
24797 { Invalid-operation~ (#1) #3 (#2) }
24798 \msg_new:nnn { fp } { unknown-type }
24799 { Unknown-type-for-~'~#1' }
24800 </code>
```

Chapter 75

l3fp-round implementation

```
24801 <*code>
24802 <@@=fp>

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
24803 \cs_new:Npn \__fp_parse_word_trunc:N
24804 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
24805 \cs_new:Npn \__fp_parse_word_floor:N
24806 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
24807 \cs_new:Npn \__fp_parse_word_ceil:N
24808 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }

(End of definition for \__fp_parse_word_trunc:N, \__fp_parse_word_floor:N, and \__fp_parse_
word_ceil:N.)

\__fp_parse_word_round:N
\__fp_parse_round:Nw
24809 \cs_new:Npn \__fp_parse_word_round:N #1#2
24810 {
24811   \__fp_parse_function:NNN
24812   \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
24813   #2
24814 }
24815 \cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest:NNN #3#4
24816 { #2 #1 #3 }

(End of definition for \__fp_parse_word_round:N and \__fp_parse_round:Nw.)
```

75.1 Rounding tools

\c__fp_five_int This is used as the half-point for which numbers are rounded up/down.

```
24817 \int_const:Nn \c__fp_five_int { 5 }
```

(End of definition for \c__fp_five_int.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in `l3fp` yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behavior (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `\__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `\__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.
- `\__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>\__fp_sep:` can expand to `0\exp_stop_f:\__fp_sep:` or `1\exp_stop_f:\__fp_sep:`.
- `\__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.

See implementation comments for details on the syntax.

```
\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \__fp_round_to_nearest_ninf:NNN
  \__fp_round_to_nearest_zero:NNN
  \__fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN
```

```
\__fp_round:NNN <final sign> <digit1> <digit2>
```

If rounding the number `<final sign><digit1>.<digit2>` to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:`. Typically used within the scope of an `\__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that `<final sign>` be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that `<final sign>` is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `\__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the `<digit2>` is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that `<digit1>` plus the result is even. In other words, round up if `<digit1>` is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```

24818 \cs_new:Npn \__fp_round_return_one:
24819 { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
24820 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
24821 {
24822   \if_meaning:w 2 #1
24823     \if_int_compare:w #3 > \c_zero_int
24824       \__fp_round_return_one:
24825     \fi:
24826   \fi:
24827   \c_zero_int
24828 }
24829 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero_int }
24830 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
24831 {
24832   \if_meaning:w 0 #1
24833     \if_int_compare:w #3 > \c_zero_int
24834       \__fp_round_return_one:
24835     \fi:
24836   \fi:
24837   \c_zero_int
24838 }
24839 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
24840 {
24841   \if_int_compare:w #3 > \c__fp_five_int
24842     \__fp_round_return_one:
24843   \else:
24844     \if_meaning:w 5 #3
24845       \if_int_odd:w #2 \exp_stop_f:
24846       \__fp_round_return_one:
24847     \fi:
24848   \fi:
24849   \fi:
24850   \c_zero_int
24851 }
24852 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
24853 {
24854   \if_int_compare:w #3 > \c__fp_five_int
24855     \__fp_round_return_one:
24856   \else:
24857     \if_meaning:w 5 #3
24858       \if_meaning:w 2 #1
24859       \__fp_round_return_one:
24860     \fi:
24861   \fi:
24862   \fi:
24863   \c_zero_int
24864 }
24865 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
24866 {
24867   \if_int_compare:w #3 > \c__fp_five_int
24868     \__fp_round_return_one:
24869   \fi:
24870   \c_zero_int
24871 }

```

```

24872 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
24873 {
24874   \if_int_compare:w #3 > \c__fp_five_int
24875     \__fp_round_return_one:
24876   \else:
24877     \if_meaning:w 5 #3
24878     \if_meaning:w 0 #1
24879     \__fp_round_return_one:
24880   \fi:
24881   \fi:
24882   \fi:
24883   \c_zero_int
24884 }
24885 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End of definition for __fp_round:NNN and others.)

__fp_round_s:NNNw

__fp_round_s:NNNw <final sign> <digit> <more digits> __fp_sep:

Similar to __fp_round:NNN, but with an extra __fp_sep:, this function expands to 0\exp_stop_f:__fp_sep: if rounding <final sign><digit>.<more digits> to an integer truncates, and to 1\exp_stop_f:__fp_sep: otherwise. The <more digits> part must be a digit, followed by something that does not overflow a \int_use:N __fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

24886 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4 \__fp_sep:
24887 {
24888   \exp_after:wN \__fp_round:NNN
24889   \exp_after:wN #1
24890   \exp_after:wN #2
24891   \int_value:w \__fp_int_eval:w
24892   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
24893   \if_meaning:w 5 #3 1 \fi:
24894   \exp_stop_f:
24895   \if_int_compare:w \__fp_int_eval:w #4 > \c_zero_int
24896     1 +
24897   \fi:
24898   \fi:
24899   #3
24900   \__fp_sep:
24901 }

```

(End of definition for __fp_round_s:NNNw.)

__fp_round_digit:Nw

\int_value:w __fp_round_digit:Nw <digit> <int expr> __fp_sep:

This function should always be called within an \int_value:w or __fp_int_eval:w expansion; it may add an extra __fp_int_eval:w, which means that the integer or integer expression should not be ended with a synonym of \relax, but with a __fp_sep: for instance.

```

24902 \cs_new:Npn \__fp_round_digit:Nw #1 #2 \__fp_sep:
24903 {
24904   \if_int_odd:w \if_meaning:w 0 #1 1 \else:
24905     \if_meaning:w 5 #1 1 \else:
24906       0 \fi: \fi: \exp_stop_f:
24907   \if_int_compare:w \__fp_int_eval:w #2 > \c_zero_int

```

```

24908         \_fp_int_eval:w 1 +
24909         \fi:
24910     \fi:
24911     #1
24912 }

```

(End of definition for _fp_round_digit:Nw.)

```

\_fp_round_neg:NNN
\_fp_round_to_nearest_neg:NNN
\_fp_round_to_nearest_ninf_neg:NNN
\_fp_round_to_nearest_zero_neg:NNN
\_fp_round_to_nearest_pinf_neg:NNN
\_fp_round_to_ninf_neg:NNN
\_fp_round_to_zero_neg:NNN
\_fp_round_to_pinf_neg:NNN

```

_fp_round_neg:NNN $\langle final\ sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle$

This expands to 0\exp_stop_f: or 1\exp_stop_f: after doing the following test. Starting from a number of the form $\langle final\ sign \rangle 0.\langle 15\ digits \rangle \langle digit_1 \rangle$ with exactly 15 (non-all-zero) digits before $\langle digit_1 \rangle$, subtract from it $\langle final\ sign \rangle 0.0\dots 0 \langle digit_2 \rangle$, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns 1\exp_stop_f:. Otherwise, i.e., if the result is rounded back to the first operand, then this function returns 0\exp_stop_f:.

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

24913 \cs_new_eq:NN \_fp_round_to_ninf_neg:NNN \_fp_round_to_pinf:NNN
24914 \cs_new:Npn \_fp_round_to_zero_neg:NNN #1 #2 #3
24915 {
24916     \if_int_compare:w #3 > \c_zero_int
24917         \_fp_round_return_one:
24918     \fi:
24919     \c_zero_int
24920 }
24921 \cs_new_eq:NN \_fp_round_to_pinf_neg:NNN \_fp_round_to_ninf:NNN
24922 \cs_new_eq:NN \_fp_round_to_nearest_neg:NNN \_fp_round_to_nearest:NNN
24923 \cs_new_eq:NN \_fp_round_to_nearest_ninf_neg:NNN
24924     \_fp_round_to_nearest_pinf:NNN
24925 \cs_new:Npn \_fp_round_to_nearest_zero_neg:NNN #1 #2 #3
24926 {
24927     \if_int_compare:w #3 < \c__fp_five_int \else:
24928         \_fp_round_return_one:
24929     \fi:
24930     \c_zero_int
24931 }
24932 \cs_new_eq:NN \_fp_round_to_nearest_pinf_neg:NNN
24933     \_fp_round_to_nearest_ninf:NNN
24934 \cs_new_eq:NN \_fp_round_neg:NNN \_fp_round_to_nearest_neg:NNN

```

(End of definition for _fp_round_neg:NNN and others.)

75.2 The round function

```

\_fp_round_o:Nw
\_fp_round_aux_o:Nw

```

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes #1 from `\_fp_round_to_nearest:NNN` to one of its analogues.

```

24935 \cs_new:Npn \_fp_round_o:Nw #1
24936 {
24937     \_fp_parse_function_all_fp_o:fnw
24938     { \_fp_round_name_from_cs:N #1 }

```

```

24939     { \_fp_round_aux_o:Nw #1 }
24940   }
24941 \cs_new:Npn \_fp_round_aux_o:Nw #1#2 @
24942 {
24943   \if_case:w
24944     \_fp_int_eval:w \_fp_array_count:n {#2} \_fp_int_eval_end:
24945     \_fp_round_no_arg_o:Nw #1 \exp:w
24946   \or: \_fp_round:Nwn #1 #2 {0} \exp:w
24947   \or: \_fp_round:Nww #1 #2 \exp:w
24948   \else: \_fp_round:Nwww #1 #2 @ \exp:w
24949   \fi:
24950   \exp_after:wN \exp_end:
24951 }

```

(End of definition for _fp_round_o:Nw and _fp_round_aux_o:Nw.)

_fp_round_no_arg_o:Nw

```

24952 \cs_new:Npn \_fp_round_no_arg_o:Nw #1
24953 {
24954   \cs_if_eq:NNTF #1 \_fp_round_to_nearest:NNN
24955   { \_fp_error_num_args:nnnn { round } { 1 } { 3 } { 0 } }
24956   {
24957     \_fp_error_num_args:ffff
24958     { \_fp_round_name_from_cs:N #1 } { 1 } { 2 } { 0 }
24959   }
24960   \exp_after:wN \c_nan_fp
24961 }

```

(End of definition for _fp_round_no_arg_o:Nw.)

_fp_round:Nwww Having three arguments is only allowed for round, not trunc, ceil, floor, so check for that case. If all is well, construct one of _fp_round_to_nearest:NNN, _fp_round_to_nearest_zero:NNN, _fp_round_to_nearest_ninf:NNN, _fp_round_to_nearest_pinf:NNN and act accordingly.

```

24962 \cs_new:Npn \_fp_round:Nwww
24963   #1#2 \_fp_sep: #3 \_fp_sep: \s__fp \_fp_chk:w #4#5#6 \_fp_sep: #7 @
24964 {
24965   \cs_if_eq:NNTF #1 \_fp_round_to_nearest:NNN
24966   {
24967     \tl_if_empty:nTF {#7}
24968     {
24969       \exp_args:Nc \_fp_round:Nww
24970       {
24971         \_fp_round_to_nearest
24972         \if_meaning:w 0 #4 _zero \else:
24973         \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
24974         :NNN
24975       }
24976       #2 \_fp_sep: #3 \_fp_sep:
24977     }
24978     {
24979       \_fp_error_num_args:ffff { round } { 1 } { 3 }
24980       { \int_eval:n { 3 + \_fp_array_count:n {#7} } }
24981       \exp_after:wN \c_nan_fp

```

```

24982     }
24983   }
24984   {
24985     \__fp_error_num_args:ffff
24986     { \__fp_round_name_from_cs:N #1 } { 1 } { 2 }
24987     { \int_eval:n { 3 + \__fp_array_count:n {#7} } }
24988     \exp_after:wN \c_nan_fp
24989   }
24990 }

```

(End of definition for __fp_round:Nwww.)

__fp_round_name_from_cs:N

```

24991 \cs_new:Npn \__fp_round_name_from_cs:N #1
24992 {
24993   \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
24994   {
24995     \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
24996     {
24997       \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
24998       { round }
24999     }
25000   }
25001 }

```

(End of definition for __fp_round_name_from_cs:N.)

__fp_round:Nww

__fp_round:Nwn

If the number of digits to round to is an integer or infinity all is good; if it is nan then just produce a nan; otherwise invalid as we have something like round(1,3.14) where the number of digits is not an integer.

__fp_round_normal:NwNNnw

__fp_round_normal:NnnwNNnn

__fp_round_pack:Nw

__fp_round_normal:NNwNnn

__fp_round_normal_end:wwNnn

__fp_round_special:NwwNnn

__fp_round_special_aux:Nw

```

25002 \cs_new:Npn \__fp_round:Nww #1#2 \__fp_sep: #3 \__fp_sep:
25003 {
25004   \__fp_small_int:wTF #3\__fp_sep: { \__fp_round:Nwn #1#2\__fp_sep: }
25005   {
25006     \if:w 3 \__fp_kind:w #3 \__fp_sep:
25007     \exp_after:wN \use_i:nn
25008     \else:
25009     \exp_after:wN \use_ii:nn
25010     \fi:
25011     { \exp_after:wN \c_nan_fp }
25012     {
25013       \__fp_invalid_operation_tl_o:ff
25014       { \__fp_round_name_from_cs:N #1 }
25015       { \__fp_array_to_clist:n { #2\__fp_sep: #3\__fp_sep: } }
25016     }
25017   }
25018 }
25019 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: #5
25020 {
25021   \if_meaning:w 1 #2
25022   \exp_after:wN \__fp_round_normal:NwNNnw
25023   \exp_after:wN #1
25024   \int_value:w #5
25025   \else:

```

```

25026         \exp_after:wN \__fp_exp_after_o:w
25027     \fi:
25028     \s__fp \__fp_chk:w #2#3#4\__fp_sep:
25029 }
25030 \cs_new:Npn \__fp_round_normal:NwNnnw #1#2 \s__fp \__fp_chk:w 1#3#4#5\__fp_sep:
25031 {
25032     \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
25033     \__fp_round_normal:NnnwNnn #5 #1 #3 {#4} {#2}
25034 }
25035 \cs_new:Npn \__fp_round_normal:NnnwNnn #1#2#3#4\__fp_sep: #5#6
25036 {
25037     \exp_after:wN \__fp_round_normal:NNwNnn
25038     \int_value:w \__fp_int_eval:w
25039     \if_int_compare:w #2 > \c_zero_int
25040         1 \int_value:w #2
25041         \exp_after:wN \__fp_round_pack:Nw
25042         \int_value:w \__fp_int_eval:w 1#3 +
25043     \else:
25044         \if_int_compare:w #3 > \c_zero_int
25045             1 \int_value:w #3 +
25046     \fi:
25047     \fi:
25048     \exp_after:wN #5
25049     \exp_after:wN #6
25050     \use_none:nnnnnn #3
25051     #1
25052     \__fp_int_eval_end:
25053     0000 0000 0000 0000 \__fp_sep: #6
25054 }
25055 \cs_new:Npn \__fp_round_pack:Nw #1
25056 { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
25057 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
25058 {
25059     \if_meaning:w 0 #2
25060         \exp_after:wN \__fp_round_special:NwNnn
25061         \exp_after:wN #1
25062     \fi:
25063     \__fp_pack_twice_four:wNNNNNNNN
25064     \__fp_pack_twice_four:wNNNNNNNN
25065     \__fp_round_normal_end:wwNnn
25066     \__fp_sep: #2
25067 }
25068 \cs_new:Npn \__fp_round_normal_end:wwNnn #1\__fp_sep:#2\__fp_sep:#3#4#5
25069 {
25070     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
25071     \__fp_sanitize:Nw #3 #4 \__fp_sep: #1 \__fp_sep:
25072 }
25073 \cs_new:Npn \__fp_round_special:NwNnn #1#2\__fp_sep:#3\__fp_sep:#4#5#6
25074 {
25075     \if_meaning:w 0 #1
25076         \__fp_case_return:nw
25077         { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
25078     \else:
25079         \exp_after:wN \__fp_round_special_aux:Nw

```

```

25080         \exp_after:wN #4
25081         \int_value:w \__fp_int_eval:w 1
25082         \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
25083     \fi:
25084     \__fp_sep:
25085 }
25086 \cs_new:Npn \__fp_round_special_aux:Nw #1#2\__fp_sep:
25087 {
25088     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
25089     \__fp_sanitiz:Nw #1#2\__fp_sep: {1000}{0000}{0000}{0000}\__fp_sep:
25090 }

```

(End of definition for __fp_round:Nww and others.)

```

25091 \code

```


Chapter 76

l3fp-parse implementation

25092 $\langle *code \rangle$
 25093 $\langle @@=fp \rangle$

76.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a $\langle floating\ point\ object \rangle$ is a floating point number or tuple. This can be extended to anything that starts with $\backslash s\_fp$ or $\backslash s\_fp\_ \langle type \rangle$ and ends with $\backslash\_fp\_sep$: with some internal structure that depends on the $\langle type \rangle$.

$\backslash\_fp\_parse:n$

$\backslash\_fp\_parse:n \{ \langle fp\ expr \rangle \}$

Evaluates the $\langle fp\ expr \rangle$ and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully f-expanded.

$\backslash\_fp\_parse_o:n$ does the same but expands once after its result.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as $\backslash int\_use:N$. Invalid tokens remaining after f-expansion lead to unrecoverable low-level T_EX errors.

(End of definition for $\backslash\_fp\_parse:n$.)

$\backslash c\_fp\_prec\_func\_int$
 $\backslash c\_fp\_prec\_hatii\_int$
 $\backslash c\_fp\_prec\_hat\_int$
 $\backslash c\_fp\_prec\_not\_int$
 $\backslash c\_fp\_prec\_juxt\_int$
 $\backslash c\_fp\_prec\_times\_int$
 $\backslash c\_fp\_prec\_plus\_int$
 $\backslash c\_fp\_prec\_comp\_int$
 $\backslash c\_fp\_prec\_and\_int$
 $\backslash c\_fp\_prec\_or\_int$
 $\backslash c\_fp\_prec\_quest\_int$
 $\backslash c\_fp\_prec\_colon\_int$
 $\backslash c\_fp\_prec\_comma\_int$
 $\backslash c\_fp\_prec\_tuple\_int$
 $\backslash c\_fp\_prec\_end\_int$

Floating point expressions are composed of numbers, given in various forms, infix operators, such as +, **, or , (which joins two numbers into a list), and prefix operators, such as the unary -, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary ** and ^ (right to left).

12 Unary +, -, ! (right to left).

11 Juxtaposition (implicit *) with no parenthesis.

- 10 Binary `*` and `/`.
- 9 Binary `+` and `-`.
- 7 Comparisons.
- 6 Logical `and`, denoted by `&&`.
- 5 Logical `or`, denoted by `||`.
- 4 Ternary operator `?:`, piece `?`.
- 3 Ternary operator `?:`, piece `:`.
- 2 Commas.
- 1 Place where a comma is allowed and generates a tuple.
- 0 Start and end of the expression.

```

25094 \int_const:Nn \c__fp_prec_func_int    { 16 }
25095 \int_const:Nn \c__fp_prec_hatii_int   { 14 }
25096 \int_const:Nn \c__fp_prec_hat_int    { 13 }
25097 \int_const:Nn \c__fp_prec_not_int    { 12 }
25098 \int_const:Nn \c__fp_prec_juxt_int    { 11 }
25099 \int_const:Nn \c__fp_prec_times_int   { 10 }
25100 \int_const:Nn \c__fp_prec_plus_int    { 9 }
25101 \int_const:Nn \c__fp_prec_comp_int    { 7 }
25102 \int_const:Nn \c__fp_prec_and_int     { 6 }
25103 \int_const:Nn \c__fp_prec_or_int      { 5 }
25104 \int_const:Nn \c__fp_prec_quest_int   { 4 }
25105 \int_const:Nn \c__fp_prec_colon_int   { 3 }
25106 \int_const:Nn \c__fp_prec_comma_int   { 2 }
25107 \int_const:Nn \c__fp_prec_tuple_int   { 1 }
25108 \int_const:Nn \c__fp_prec_end_int     { 0 }

```

(End of definition for `\c__fp_prec_func_int` and others.)

76.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to `f-expand` tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets

us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w (stuff)
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a `\__fp_sep:` in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 \__fp_sep:
\exp:w \exp_end: 333444 \__fp_sep:
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `\__fp_sep:`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 \__fp_sep: 333444 \__fp_sep:
```

which can safely perform the addition by grabbing two arguments delimited by `\__fp_sep:`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN \__fp_sep:` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `\__fp..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

76.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because $\times$ has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation

$41-2^3*4+5$. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c__fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have $41-8*4+5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8*4 = 32$.
- We now have $41-32+5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have $9+5$.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `\__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle \textit{precedence} \rangle$ the argument of `\__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `\__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle \textit{number} \rangle$, performing no computation, and finds the following binary $\langle \textit{operator} \rangle$. Then it expands to

```

<number>
  \_fp_parse_infix_<operator>:N <precedence>

```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `\_fp_parse_infix_<operator>:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `<precedence>` (of the earlier operator) to the `infix` auxiliary for the following `<operator>`, to know whether to perform the computation of the `<operator>`. If it should not be performed, the `infix` auxiliary expands to

```

@ \use_none:n \_fp_parse_infix_<operator>:N

```

and otherwise it calls `\_fp_parse_operand:Nw` with the precedence of the `<operator>` to find its second operand `<number2>` and the next `<operator2>`, and expands to

```

@ \_fp_parse_apply_binary:NwNwN
  <operator> <number2>
@ \_fp_parse_infix_<operator2>:N

```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `<number>` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\_fp_parse_operand:Nw <precedence>` with some of the expansion control removed is

```

\exp_after:wN \_fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
  \_fp_parse_one:Nw <precedence>

```

This expands `\_fp_parse_one:Nw <precedence>` completely, which finds a number, wraps the next `<operator>` into an `infix` function, feeds this function the `<precedence>`, and expands it, yielding either

```

\_fp_parse_continue:NwN <precedence>
<number> @
\use_none:n \_fp_parse_infix_<operator>:N

```

or

```

\_fp_parse_continue:NwN <precedence>
<number> @
\_fp_parse_apply_binary:NwNwN
  <operator> <number2>
@ \_fp_parse_infix_<operator2>:N

```

The definition of `\_fp_parse_continue:NwN` is then very simple:

```

\cs_new:Npn \_fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }

```

In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then `<number> @ \__fp_parse_infix_<operator>:N`. In the second case, #3 is `\__fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number2>` and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
  <precedence> <number> @
  <operator> <number2>
@ \__fp_parse_infix_<operator2>:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\__fp_<operator>_o:ww <number> <number2>
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_<operator2>:N <precedence>
```

where `\__fp_<operator>_o:ww` computes `<number> <operator> <number2>` and expands after the result, thus triggers the comparison of the precedence of the `<operator2>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

76.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `\__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `\__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `\__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `\__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `\__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is then

parsed as $0 > -(2+3)$: the addition is performed because it binds more tightly than the comparison which precedes $-$. The correct approach is for a unary $-$ to perform operations whose precedence is greater than both that of the previous operation, and that of the unary $-$ itself. The unary $-$ is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `\_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous `<precedence>` to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

76.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form `<significand>e<exponent>`, where the `<significand>` is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “`e<exponent>`” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs $+$ or $-$ and a non-empty string of decimal digits. The `<significand>` can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the `<exponent>` can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from c-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the `<significand>` of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `\_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `\_fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `\__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\__fp_infix_⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `#1` lies in [65, 90] (uppercase letters) or [97, 112] (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3, 6, 7, 8, 11, 12} should work without trouble, but not {1, 2, 4, 10, 13}, and of course {0, 5, 9} cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back

in the input stream, then **f**-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the **f**-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The **f**-expansion is performed by `\__fp_parse_expand:w`.

76.2 Main auxiliary functions

`\__fp_parse_operand:Nw` `\exp:w \__fp_parse_operand:Nw <precedence> \__fp_parse_expand:w`
 Reads the "...", performing every computation with a precedence higher than `<precedence>`, then expands to

`<result> @ \__fp_parse_infix_<operation>:N ...`

where the `<operation>` is the first operation with a lower precedence, possibly `end`, and the "..." start just after the `<operation>`.

(End of definition for __fp_parse_operand:Nw.)

`\__fp_parse_infix_+:N` `\__fp_parse_infix_+:N <precedence> ...`
 If `+` has a precedence higher than the `<precedence>`, cleans up a second `<operand>` and finds the `<operation2 which follows, and expands to`

`@ __fp_parse_apply_binary:NwNwN + <operand> @ __fp_parse_infix_<operation2
...`

Otherwise expands to

`@ \use_none:n \__fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(End of definition for __fp_parse_infix_+:N.)

`\__fp_parse_one:Nw` `\__fp_parse_one:Nw <precedence> ...`
 Cleans up one or two operands depending on how the precedence of the next operation compares to the `<precedence>`. If the following `<operation>` has a precedence higher than `<precedence>`, expands to

`<operand1> @ \__fp_parse_apply_binary:NwNwN <operation> <operand2> @`
`\__fp_parse_infix_<operation2`

and otherwise expands to

`<operand> @ \use_none:n \__fp_parse_infix_<operation>:N ...`

(End of definition for __fp_parse_one:Nw.)

76.3 Helpers

`\__fp_parse_expand:w` `\exp:w \__fp_parse_expand:w <tokens>`

This function must always come within a `\exp:w` expansion. The `<tokens>` should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
25109 \cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(End of definition for __fp_parse_expand:w.)

`\__fp_parse_return_sep:w` This very odd function swaps its position with the following `\fi:` and removes `\__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

```
25110 \cs_new:Npn \__fp_parse_return_sep:w
25111     #1 \fi: \__fp_parse_expand:w { \fi: \__fp_sep: #1 }
```

(End of definition for __fp_parse_return_sep:w.)

`\__fp_parse_digits_vii:N` These functions must be called within an `\int_value:w` or `\__fp_int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

```
\__fp_parse_digits_vii:N    <digits> \__fp_sep: <filling 0> \__fp_sep: <length>
\__fp_parse_digits_vi:N
\__fp_parse_digits_v:N
\__fp_parse_digits_iv:N
\__fp_parse_digits_iii:N
\__fp_parse_digits_ii:N
\__fp_parse_digits_i:N
\__fp_parse_digits_:N
```

where `<filling 0>` is a string of zeros such that `<digits> <filling 0>` has the length given by the index of the function, and `<length>` is the number of zeros in the `<filling 0>` string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```
25112 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
25113 {
25114     \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
25115     {
25116         \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
25117         \token_to_str:N ##1 \exp_after:wN #2 \exp:w
25118         \else:
25119             \__fp_parse_return_sep:w #3 ##1
25120         \fi:
25121         \__fp_parse_expand:w
25122     }
25123 }
25124 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 \__fp_sep: 7 }
25125 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 \__fp_sep: 6 }
25126 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 \__fp_sep: 5 }
25127 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 0000 \__fp_sep: 4 }
25128 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 000 \__fp_sep: 3 }
25129 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 00 \__fp_sep: 2 }
25130 \__fp_tmp:w {i} \__fp_parse_digits_:N { 0 \__fp_sep: 1 }
25131 \cs_new:Npn \__fp_parse_digits_:N { \__fp_sep: \__fp_sep: 0 }
```

(End of definition for __fp_parse_digits_vii:N and others.)

76.4 Parsing one number

`\__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `\__fp_parse_infix_...` csname. #1 is the previous $\langle precedence \rangle$, and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier `f`-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the $\text{\LaTeX 2}_{\epsilon}$ command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `\__fp_parse_one_fp:NN` which deals with it robustly.

```

25132 \cs_new:Npn \__fp_parse_one:Nw #1 #2
25133 {
25134   \if_catcode:w \scan_stop: \exp_not:N #2
25135   \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
25136     \exp_after:wN \reverse_if:N
25137   \fi:
25138   \if_meaning:w \scan_stop: #2
25139     \exp_after:wN \exp_after:wN
25140     \exp_after:wN \__fp_parse_one_fp:NN
25141   \else:
25142     \exp_after:wN \exp_after:wN
25143     \exp_after:wN \__fp_parse_one_register:NN
25144   \fi:
25145   \else:
25146     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
25147       \exp_after:wN \exp_after:wN
25148       \exp_after:wN \__fp_parse_one_digit:NN
25149     \else:
25150       \exp_after:wN \exp_after:wN
25151       \exp_after:wN \__fp_parse_one_other:NN
25152     \fi:
25153   \fi:
25154   #1 #2
25155 }
```

(End of definition for `\__fp_parse_one:Nw`.)

`\__fp_parse_one_fp:NN` This function receives a $\langle precedence \rangle$ and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

`\_fp_exp_after_expr_mark_f:nw`
`\__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `\__fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_expr_mark` is a premature end, we call `\__fp_exp_after_expr_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s__fp`, we call `\__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_⟨type⟩` and defining `\__fp_exp_after_⟨type⟩_f:nw`. In all cases, we make sure that the second argument of `\__fp_parse_infix:NN` is correctly expanded. A special case only enabled in $\text{\LaTeX 2}_{\epsilon}$ is that if `\protect` is encountered then

the error message mentions the control sequence which follows it rather than `\protect` itself. The test for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}_{2\epsilon}$ uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

25156 \cs_new:Npn \__fp_parse_one_fp:NN #1
25157 {
25158   \__fp_exp_after_any_f:nw
25159   {
25160     \exp_after:wN \__fp_parse_infix:NN
25161     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
25162   }
25163 }
25164 \cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
25165 {
25166   \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
25167   {
25168     \c__fp_prec_comma_int { }
25169     \c__fp_prec_tuple_int { }
25170     \c__fp_prec_end_int
25171     {
25172       \exp_after:wN \c__fp_empty_tuple_fp
25173       \exp:w \exp_end_continue_f:w
25174     }
25175   }
25176   {
25177     \msg_expandable_error:nn { fp } { early-end }
25178     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
25179   }
25180   #1
25181 }
25182 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
25183 {
25184   \msg_expandable_error:nnn { kernel } { bad-variable }
25185   {#2}
25186   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
25187 }
25188 \cs_set_protected:Npn \__fp_tmp:w #1
25189 {
25190   \cs_if_exist:NT #1
25191   {
25192     \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
25193     {
25194       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
25195       \str_if_eq:nnTF {##2} { \protect }
25196       {
25197         \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
25198         {
25199           \msg_expandable_error:nnn { fp }
25200           { robust-cmd }
25201         }
25202       }
25203     }
25204     \msg_expandable_error:nnn { kernel }
25205     { bad-variable } {##2}
25206   }

```

```

25207     }
25208   }
25209 }
25210 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }

```

(End of definition for __fp_parse_one_fp:NN, __fp_exp_after_expr_mark_f:nw, and __fp_exp_after_?_f:nw.)

```

\__fp_parse_one_register:NN
  \__fp_parse_one_register_aux:Nw
  \__fp_parse_one_register_auxii:wwwNw
  \__fp_parse_one_register_int:www
  \__fp_parse_one_register_mu:www
  \__fp_parse_one_register_dim:ww

```

This is called whenever #2 is a control sequence other than \scan_stop: in meaning. We special-case \wd, \ht, \dp (see later) and otherwise assume that it is a register, but carefully unpack it with \tex_the:D within braces. First, we find the exponent following #2. Then we unpack #2 with \tex_the:D, and the auxii auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of pt. For integers, simply convert $\langle value \rangle e \langle exponent \rangle$ to a floating point number with __fp_parse:n (this is somewhat wasteful). For other registers, the decimal rounding provided by T_EX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with \int_value:w \dim_to_decimal_in_sp:n { $\langle decimal value \rangle$ pt }, and use an auxiliary of \dim_to_fp:n, which performs the multiplication by 2^{-16} , correctly rounded.

```

25211 \cs_new:Npn \__fp_parse_one_register:NN #1#2
25212 {
25213   \exp_after:wN \__fp_parse_infix_after_operand:NwN
25214   \exp_after:wN #1
25215   \exp:w \exp_end_continue_f:w
25216   \__fp_parse_one_register_special:N #2
25217   \exp_after:wN \__fp_parse_one_register_aux:Nw
25218   \exp_after:wN #2
25219   \int_value:w
25220   \exp_after:wN \__fp_parse_exponent:N
25221   \exp:w \__fp_parse_expand:w
25222 }
25223 \cs_new:Npe \__fp_parse_one_register_aux:Nw #1
25224 {
25225   \exp_not:n
25226   {
25227     \exp_after:wN \use:nn
25228     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
25229   }
25230   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
25231   \__fp_sep: \exp_not:N \__fp_parse_one_register_dim:ww
25232   \tl_to_str:n { pt } \__fp_sep: \exp_not:N \__fp_parse_one_register_mu:www
25233   . \tl_to_str:n { pt } \__fp_sep: \exp_not:N \__fp_parse_one_register_int:www
25234   \s__fp_stop
25235 }
25236 \exp_args:Nno \use:nn
25237 { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
25238 { \tl_to_str:n { pt } #3 \__fp_sep: #4#5 \s__fp_stop }
25239 { #4 #1.#2\__fp_sep: }
25240 \exp_args:Nno \use:nn
25241 { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
25242 { \tl_to_str:n { mu } \__fp_sep: #2 \__fp_sep: }
25243 { \__fp_parse_one_register_dim:ww #1 \__fp_sep: }
25244 \cs_new:Npn \__fp_parse_one_register_int:www #1\__fp_sep: #2.\__fp_sep: #3\__fp_sep:
25245 { \__fp_parse:n { #1 e #3 } }

```

```

25246 \cs_new:Npn \__fp_parse_one_register_dim:ww #1\__fp_sep: #2\__fp_sep:
25247 {
25248   \exp_after:wN \__fp_from_dim_test:ww
25249   \int_value:w #2 \exp_after:wN ,
25250   \int_value:w \dim_to_decimal_in_sp:n { #1 pt } \__fp_sep:
25251 }

```

(End of definition for __fp_parse_one_register:NN and others.)

```

\__fp_parse_one_register_special:N
\__fp_parse_one_register_math:NNw
  \__fp_parse_one_register_wd:w
  \__fp_parse_one_register_wd:Nw

```

The \wd, \dp, \ht primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker e. Once that “exponent” is found, use \tex_the:D to find the box dimension and then copy what we did for dimensions.

```

25252 \cs_new:Npn \__fp_parse_one_register_special:N #1
25253 {
25254   \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
25255   \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
25256   \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
25257   \if_meaning:w \infty #1
25258     \__fp_parse_one_register_math:NNw \infty #1
25259   \fi:
25260   \if_meaning:w \pi #1
25261     \__fp_parse_one_register_math:NNw \pi #1
25262   \fi:
25263 }
25264 \cs_new:Npn \__fp_parse_one_register_math:NNw
25265   #1#2#3#4 \__fp_parse_expand:w
25266 {
25267   #3
25268   \str_if_eq:nnTF {#1} {#2}
25269   {
25270     \msg_expandable_error:nnn
25271       { fp } { infinity-pi } {#1}
25272     \c_nan_fp
25273   }
25274   { #4 \__fp_parse_expand:w }
25275 }
25276 \cs_new:Npn \__fp_parse_one_register_wd:w
25277   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
25278 {
25279   #1
25280   \exp_after:wN \__fp_parse_one_register_wd:Nw
25281   #4 \__fp_parse_expand:w e
25282 }
25283 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 \__fp_sep:
25284 {
25285   \exp_after:wN \__fp_from_dim_test:ww
25286   \exp_after:wN 0 \exp_after:wN ,
25287   \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } \__fp_sep:
25288 }

```

(End of definition for __fp_parse_one_register_special:N and others.)

```

\__fp_parse_one_digit:NN

```

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with __fp_sanitizewN,

then `\__fp_parse_infix_after_operand:NwN` expands `\__fp_parse_infix:NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

25289 \cs_new:Npn \__fp_parse_one_digit:NN #1
25290 {
25291   \exp_after:wN \__fp_parse_infix_after_operand:NwN
25292   \exp_after:wN #1
25293   \exp:w \exp_end_continue_f:w
25294   \exp_after:wN \__fp_sanitize:wN
25295   \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
25296 }

```

(End of definition for `\__fp_parse_one_digit:NN`.)

`\__fp_parse_one_other:NN` For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `\__fp_parse_letters:N` beyond this one and give the result to `\__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `\__fp_parse_prefix_{operator}:Nw`.

```

25297 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
25298 {
25299   \if_int_compare:w
25300     \__fp_int_eval:w
25301     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
25302     = 3 \exp_stop_f:
25303     \exp_after:wN \__fp_parse_word:Nw
25304     \exp_after:wN #1
25305     \exp_after:wN #2
25306     \exp:w \exp_after:wN \__fp_parse_letters:N
25307     \exp:w
25308   \else:
25309     \exp_after:wN \__fp_parse_prefix:NNN
25310     \exp_after:wN #1
25311     \exp_after:wN #2
25312     \cs:w
25313     __fp_parse_prefix_ \token_to_str:N #2 :Nw
25314     \exp_after:wN
25315     \cs_end:
25316     \exp:w
25317   \fi:
25318   \__fp_parse_expand:w
25319 }

```

(End of definition for `\__fp_parse_one_other:NN`.)

`\__fp_parse_word:Nw` Finding letters is a simple recursion. Once `\__fp_parse_letters:N` has done its job, `\__fp_parse_letters:N` we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

25320 \cs_new:Npn \__fp_parse_word:Nw #1#2\__fp_sep:
25321 {
25322   \cs_if_exist_use:cF { __fp_parse_word_#2:N }
25323   {
25324     \cs_if_exist_use:cF
25325     { __fp_parse_caseless_ \str_casefold:n {#2} :N }
25326     {
25327       \msg_expandable_error:nnn
25328       { fp } { unknown-fp-word } {#2}
25329       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
25330       \__fp_parse_infix:NN
25331     }
25332   }
25333   #1
25334 }
25335 \cs_new:Npn \__fp_parse_letters:N #1
25336 {
25337   \exp_end_continue_f:w
25338   \if_int_compare:w
25339   \if_catcode:w \scan_stop: \exp_not:N #1
25340   0
25341   \else:
25342     \__fp_int_eval:w
25343     ( ' #1 \if_int_compare:w ' #1 > ' Z - 32 \fi: ) / 26
25344     \fi:
25345     = 3 \exp_stop_f:
25346     \exp_after:wN #1
25347     \exp:w \exp_after:wN \__fp_parse_letters:N
25348     \exp:w
25349   \else:
25350     \__fp_parse_return_sep:w #1
25351     \fi:
25352     \__fp_parse_expand:w
25353 }

```

(End of definition for __fp_parse_word:Nw and __fp_parse_letters:N.)

__fp_parse_prefix:NNN
 __fp_parse_prefix_unknown:NNN

For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from __fp_parse_one:Nw.

```

25354 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
25355 {
25356   \if_meaning:w \scan_stop: #3
25357   \exp_after:wN \__fp_parse_prefix_unknown:NNN
25358   \exp_after:wN #2
25359   \fi:
25360   #3 #1
25361 }
25362 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
25363 {

```



```

25364 \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
25365 {
25366   \msg_expandable_error:nnn
25367   { fp } { missing-number } {#1}
25368   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
25369   \__fp_parse_infix:NN #3 #1
25370 }
25371 {
25372   \msg_expandable_error:nnn
25373   { fp } { unknown-symbol } {#1}
25374   \__fp_parse_one:Nw #3
25375 }
25376 }

```

(End of definition for __fp_parse_prefix:NNN and __fp_parse_prefix_unknown:NNN.)

76.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions __fp_parse_large...; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle \text{exp}_1 \rangle < 0$, then read the significand with the set of functions __fp_parse_small... Once the significand is read, read the exponent if *e* is present.

__fp_parse_trim_zeros:N This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call __fp_parse_large:N (the significand is ≥ 1); if it is ., then continue trimming zeros with __fp_parse_strim_zeros:N; otherwise, our number is exactly zero, and we call __fp_parse_zero: to take care of that case.

```

25377 \cs_new:Npn \__fp_parse_trim_zeros:N #1
25378 {
25379   \if:w 0 \exp_not:N #1
25380     \exp_after:wN \__fp_parse_trim_zeros:N
25381     \exp:w
25382   \else:
25383     \if:w . \exp_not:N #1
25384       \exp_after:wN \__fp_parse_strim_zeros:N
25385       \exp:w
25386     \else:
25387       \__fp_parse_trim_end:w #1
25388     \fi:
25389   \fi:
25390   \__fp_parse_expand:w
25391 }
25392 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
25393 {
25394   \fi:
25395   \fi:
25396   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25397     \exp_after:wN \__fp_parse_large:N
25398   \else:
25399     \exp_after:wN \__fp_parse_zero:

```

```

25400     \fi:
25401     #1
25402   }

```

(End of definition for `\__fp_parse_trim_zeros:N` and `\__fp_parse_trim_end:w`.)

`\__fp_parse_strim_zeros:N` If we have removed all digits until a period (or if the body started with a period), then enter the “`small_trim`” loop which outputs `-1` for each removed 0. Those `-1` are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\__fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

25403 \cs_new:Npn \__fp_parse_strim_zeros:N #1
25404 {
25405   \if:w 0 \exp_not:N #1
25406     - 1
25407     \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
25408   \else:
25409     \__fp_parse_strim_end:w #1
25410   \fi:
25411   \__fp_parse_expand:w
25412 }
25413 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
25414 {
25415   \fi:
25416   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25417     \exp_after:wN \__fp_parse_small:N
25418   \else:
25419     \exp_after:wN \__fp_parse_zero:
25420   \fi:
25421   #1
25422 }

```

(End of definition for `\__fp_parse_strim_zeros:N` and `\__fp_parse_strim_end:w`.)

`\__fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `\__fp-sanitize:wN`, which removes everything and leaves an exact zero.

```

25423 \cs_new:Npn \__fp_parse_zero:
25424 {
25425   \exp_after:wN \__fp_sep: \exp_after:wN 1
25426   \int_value:w \__fp_parse_exponent:N
25427 }

```

(End of definition for `\__fp_parse_zero:.`)

76.4.2 Number: small significand

`\__fp_parse_small:N` This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `\__fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the

`pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

25428 \cs_new:Npn \__fp_parse_small:N #1
25429 {
25430   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
25431   \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
25432   \exp_after:wN \__fp_parse_small_leading:wwNN
25433   \int_value:w 1
25434   \exp_after:wN \__fp_parse_digits_vii:N
25435   \exp:w \__fp_parse_expand:w
25436 }

```

(End of definition for `\__fp_parse_small:N`.)

```

\__fp_parse_small_leading:wwNN
\__fp_parse_small_leading:wwNN 1 <digits> \__fp_sep: <zeros> \__fp_sep:
<number of zeros>

```

We leave `<digits>` `<zeros>` in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

25437 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 \__fp_sep: #2\__fp_sep: #3 #4
25438 {
25439   #1 #2
25440   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
25441   \exp_after:wN 0
25442   \int_value:w \__fp_int_eval:w 1
25443   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
25444     \token_to_str:N #4
25445     \exp_after:wN \__fp_parse_small_trailing:wwNN
25446     \int_value:w 1
25447     \exp_after:wN \__fp_parse_digits_vi:N
25448     \exp:w
25449   \else:
25450     0000 0000 \__fp_parse_exponent:Nw #4
25451   \fi:
25452   \__fp_parse_expand:w
25453 }

```

(End of definition for `\__fp_parse_small_leading:wwNN`.)

```

\__fp_parse_small_trailing:wwNN
\__fp_parse_small_trailing:wwNN 1 <digits> \__fp_sep: <zeros> \__fp_sep:
<number of zeros> <next token>

```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the `<next token>` is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

25454 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 \__fp_sep: #2\__fp_sep: #3 #4
25455 {
25456   #1 #2
25457   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
25458     \token_to_str:N #4

```

```

25459     \exp_after:wN \__fp_parse_small_round:NN
25460     \exp_after:wN #4
25461     \exp:w
25462   \else:
25463     0 \__fp_parse_exponent:Nw #4
25464   \fi:
25465   \__fp_parse_expand:w
25466 }

```

(End of definition for `\__fp_parse_small_trailing:wwNN`.)

```

\__fp_parse_pack_trailing:NNNNNNww
\__fp_parse_pack_leading:NNNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `\__fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

25467 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww
25468   #1 #2 #3#4#5#6 #7\__fp_sep: #8 \__fp_sep:
25469   {
25470     \if_meaning:w 2 #2 + 1 \fi:
25471     \__fp_sep: #8 + #1 \__fp_sep: {#3#4#5#6} {#7}\__fp_sep:
25472   }
25473 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6\__fp_sep: #7\__fp_sep:
25474   {
25475     + #7
25476     \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
25477     \__fp_sep: 0 {#2#3#4#5} {#6}
25478   }
25479 \cs_new:Npn \__fp_parse_pack_carry:w \fi: \__fp_sep: 0 #1
25480   { \fi: + 1 \__fp_sep: 0 {1000} }

```

(End of definition for `\__fp_parse_pack_trailing:NNNNNNww`, `\__fp_parse_pack_leading:NNNNNNww`, and `\__fp_parse_pack_carry:w`.)

76.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

```

\__fp_parse_large:N

```

This function is followed by the first non-zero digit of a “large” significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

25481 \cs_new:Npn \__fp_parse_large:N #1
25482   {
25483     \exp_after:wN \__fp_parse_large_leading:wwNN
25484     \int_value:w 1 \token_to_str:N #1

```

```

25485     \exp_after:wN \__fp_parse_digits_vii:N
25486     \exp:w \__fp_parse_expand:w
25487 }

```

(End of definition for __fp_parse_large:N.)

```

\__fp_parse_large_leading:wwNN    \__fp_parse_large_leading:wwNN 1 <digits> \__fp_sep: <zeros> \__fp_sep:
                                   <number of zeros> <next token>

```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the *<number of zeros>* (number of digits missing). Then prepare to pack the 8 first digits. If the *<next token>* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the *small* functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *<zeros>* to complete the 8 first digits, insert 8 more, and look for an exponent.

```

25488 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 \__fp_sep: #2\__fp_sep: #3 #4
25489 {
25490   + \c__fp_half_prec_int - #3
25491   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
25492   \int_value:w \__fp_int_eval:w 1 #1
25493   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
25494     \exp_after:wN \__fp_parse_large_trailing:wwNN
25495     \int_value:w 1 \token_to_str:N #4
25496     \exp_after:wN \__fp_parse_digits_vi:N
25497     \exp:w
25498   \else:
25499     \if:w . \exp_not:N #4
25500       \exp_after:wN \__fp_parse_small_leading:wwNN
25501       \int_value:w 1
25502       \cs:w
25503         __fp_parse_digits_
25504         \__fp_int_to_roman:w #3
25505         :N \exp_after:wN
25506       \cs_end:
25507       \exp:w
25508     \else:
25509       #2
25510       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
25511       \exp_after:wN 0
25512       \int_value:w 1 0000 0000
25513       \__fp_parse_exponent:Nw #4
25514     \fi:
25515   \fi:
25516   \__fp_parse_expand:w
25517 }

```

(End of definition for __fp_parse_large_leading:wwNN.)

```

\__fp_parse_large_trailing:wwNN    \__fp_parse_large_trailing:wwNN 1 <digits> \__fp_sep: <zeros> \__fp_sep:
                                   <number of zeros> <next token>

```

We have just read 15 digits. If the *<next token>* is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the *pack_trailing* function. We keep the *<digits>* and this 16-th digit, and find how this should be rounded using *__fp_parse_large_round:NN*. Otherwise, the exponent shift is the number of

$\langle \text{digits} \rangle$, 7 minus the $\langle \text{number of zeros} \rangle$, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate small auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the $\langle \text{zeros} \rangle$ and providing a 16-th digit of 0.

```

25518 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 \__fp_sep: #2\__fp_sep: #3 #4
25519 {
25520   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
25521     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
25522     \exp_after:wN \c__fp_half_prec_int
25523     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
25524     \exp_after:wN \__fp_parse_large_round:NN
25525     \exp_after:wN #4
25526     \exp:w
25527   \else:
25528     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
25529     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
25530     \int_value:w \__fp_int_eval:w 1 #1
25531     \if:w . \exp_not:N #4
25532       \exp_after:wN \__fp_parse_small_trailing:wwNN
25533       \int_value:w 1
25534       \cs:w
25535         __fp_parse_digits_
25536         \__fp_int_to_roman:w #3
25537         :N \exp_after:wN
25538       \cs_end:
25539       \exp:w
25540     \else:
25541       #2 0 \__fp_parse_exponent:Nw #4
25542     \fi:
25543   \fi:
25544   \__fp_parse_expand:w
25545 }

```

(End of definition for `\__fp_parse_large_trailing:wwNN`.)

76.4.4 Number: beyond 16 digits, rounding

`\__fp_parse_round_loop:N` This loop is called when rounding a number (whether the mantissa is small or large).
`\__fp_parse_round_up:N` It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `\__fp_sep:0`, otherwise by `\__fp_sep:1`. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

25546 \cs_new:Npn \__fp_parse_round_loop:N #1
25547 {
25548   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25549     + 1
25550     \if:w 0 \token_to_str:N #1
25551       \exp_after:wN \__fp_parse_round_loop:N
25552       \exp:w
25553     \else:
25554       \exp_after:wN \__fp_parse_round_up:N

```

```

25555         \exp:w
25556         \fi:
25557     \else:
25558         \__fp_parse_return_sep:w 0 #1
25559         \fi:
25560         \__fp_parse_expand:w
25561     }
25562 \cs_new:Npn \__fp_parse_round_up:N #1
25563 {
25564     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25565         + 1
25566         \exp_after:wN \__fp_parse_round_up:N
25567         \exp:w
25568     \else:
25569         \__fp_parse_return_sep:w 1 #1
25570         \fi:
25571         \__fp_parse_expand:w
25572     }

```

(End of definition for __fp_parse_round_loop:N and __fp_parse_round_up:N.)

__fp_parse_round_after:wN After the loop __fp_parse_round_loop:N, this function fetches an exponent with __fp_parse_exponent:N, and combines it with the number of digits counted by __fp_parse_round_loop:N. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

25573 \cs_new:Npn \__fp_parse_round_after:wN #1\__fp_sep: #2
25574 {
25575     + #2 \exp_after:wN \__fp_sep:
25576     \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
25577 }

```

(End of definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to __fp_sep:<exponent> only. Otherwise, we expand to +0 or +1, then __fp_sep:<exponent>. To decide which, call __fp_round_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```

25578 \cs_new:Npn \__fp_parse_small_round:NN #1#2
25579 {
25580     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
25581         +
25582         \exp_after:wN \__fp_round_s:NNNw
25583         \exp_after:wN 0
25584         \exp_after:wN #1
25585         \exp_after:wN #2
25586         \int_value:w \__fp_int_eval:w
25587         \exp_after:wN \__fp_parse_round_after:wN
25588         \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
25589         \exp_after:wN \__fp_parse_round_loop:N

```

```

25590         \exp:w
25591     \else:
25592         \__fp_parse_exponent:Nw #2
25593     \fi:
25594     \__fp_parse_expand:w
25595 }

```

(End of definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.)

```

\__fp_parse_large_round:NN
  \__fp_parse_large_round_test:NN
  \__fp_parse_large_round_aux:wNN

```

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

25596 \cs_new:Npn \__fp_parse_large_round:NN #1#2
25597 {
25598     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
25599     +
25600     \exp_after:wN \__fp_round_s:NNNw
25601     \exp_after:wN 0
25602     \exp_after:wN #1
25603     \exp_after:wN #2
25604     \int_value:w \__fp_int_eval:w
25605     \exp_after:wN \__fp_parse_large_round_aux:wNN
25606     \int_value:w \__fp_int_eval:w 1
25607     \exp_after:wN \__fp_parse_round_loop:N
25608 \else: %^^A could be dot, or e, or other
25609     \exp_after:wN \__fp_parse_large_round_test:NN
25610     \exp_after:wN #1
25611     \exp_after:wN #2
25612 \fi:
25613 }
25614 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
25615 {
25616     \if:w . \exp_not:N #2
25617         \exp_after:wN \__fp_parse_small_round:NN
25618         \exp_after:wN #1
25619         \exp:w
25620     \else:
25621         \__fp_parse_exponent:Nw #2
25622     \fi:
25623     \__fp_parse_expand:w
25624 }
25625 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 \__fp_sep: #2 #3
25626 {
25627     + #2
25628     \exp_after:wN \__fp_parse_round_after:wN
25629     \int_value:w \__fp_int_eval:w #1
25630     \if:w . \exp_not:N #3
25631         + 0 * \__fp_int_eval:w 0

```



```

25632         \exp_after:wN \__fp_parse_round_loop:N
25633         \exp:w \exp_after:wN \__fp_parse_expand:w
25634     \else:
25635         \exp_after:wN \__fp_sep:
25636         \exp_after:wN 0
25637         \exp_after:wN #3
25638     \fi:
25639 }

```

(End of definition for `\__fp_parse_large_round:NN`, `\__fp_parse_large_round_test:NN`, and `\__fp_parse_large_round_aux:wNN`.)

76.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading `3.2`, `\c_pi_fp` is expanded to `\s__fp \__fp_chk:w 1 0 {-1} {3141}` ... `\__fp_sep:` and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `\__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`\__fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `\__fp_int_eval:w ...` there if needed.

```

25640 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
25641 {
25642     \exp_after:wN \__fp_sep:
25643     \int_value:w #2 \__fp_parse_exponent:N #1
25644 }

```

(End of definition for `\__fp_parse_exponent:Nw`.)

`\__fp_parse_exponent:N` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a `\__fp_sep:.` If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent,

less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

25645 \cs_new:Npn \__fp_parse_exponent:N #1
25646 {
25647   \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
25648     \exp_after:wN \__fp_parse_exponent_aux:NN
25649     \exp_after:wN #1
25650     \exp:w
25651   \else:
25652     0 \__fp_parse_return_sep:w #1
25653   \fi:
25654   \__fp_parse_expand:w
25655 }
25656 \cs_new:Npn \__fp_parse_exponent_aux:NN #1#2
25657 {
25658   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
25659     0 \else: '#2 \fi: > '9 \exp_stop_f:
25660     0 \exp_after:wN \__fp_sep: \exp_after:wN #1
25661   \else:
25662     \exp_after:wN \__fp_parse_exponent_sign:N
25663   \fi:
25664   #2
25665 }

```

(End of definition for __fp_parse_exponent:N and __fp_parse_exponent_aux:NN.)

__fp_parse_exponent_sign:N Read signs one by one (if there is any).

```

25666 \cs_new:Npn \__fp_parse_exponent_sign:N #1
25667 {
25668   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
25669     \exp_after:wN \__fp_parse_exponent_sign:N
25670     \exp:w \exp_after:wN \__fp_parse_expand:w
25671   \else:
25672     \exp_after:wN \__fp_parse_exponent_body:N
25673     \exp_after:wN #1
25674   \fi:
25675 }

```

(End of definition for __fp_parse_exponent_sign:N.)

__fp_parse_exponent_body:N An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

25676 \cs_new:Npn \__fp_parse_exponent_body:N #1
25677 {
25678   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25679     \token_to_str:N #1
25680     \exp_after:wN \__fp_parse_exponent_digits:N
25681     \exp:w
25682   \else:
25683     \__fp_parse_exponent_keep:NTF #1
25684     { \__fp_parse_return_sep:w #1 }
25685     {
25686       \exp_after:wN \__fp_sep:
25687       \exp:w

```

```

25688     }
25689     \fi:
25690     \__fp_parse_expand:w
25691 }

```

(End of definition for __fp_parse_exponent_body:N.)

__fp_parse_exponent_digits:N Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a __fp_sep:. Note that we do not check for overflow of the exponent, hence there can be a T_EX error. It is mostly harmless, except when parsing 0e9876543210, which should be a valid representation of 0, but is not.

```

25692 \cs_new:Npn \__fp_parse_exponent_digits:N #1
25693 {
25694   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
25695     \token_to_str:N #1
25696     \exp_after:wN \__fp_parse_exponent_digits:N
25697     \exp:w
25698   \else:
25699     \__fp_parse_return_sep:w #1
25700   \fi:
25701   \__fp_parse_expand:w
25702 }

```

(End of definition for __fp_parse_exponent_digits:N.)

__fp_parse_exponent_keep:N This is the last building block for parsing exponents. The argument #1 is already fully expanded, and neither + nor - nor a digit. It can be:

- \s__fp, marking the start of an internal floating point, invalid here;
- another control sequence equal to \relax, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than +, - or digits, again, an error.

```

25703 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
25704 {
25705   \if_catcode:w \scan_stop: \exp_not:N #1
25706     \if_meaning:w \scan_stop: #1
25707       \if:w 0 \__fp_str_if_eq:nn { \s__fp } { \exp_not:N #1 }
25708       0
25709       \msg_expandable_error:nnn
25710         { fp } { after-e } { floating-point~ }
25711       \prg_return_true:
25712     \else:
25713       0
25714       \msg_expandable_error:nnn
25715         { kernel } { bad-variable } { #1 }
25716       \prg_return_false:
25717     \fi:
25718   \else:
25719     \if:w 0 \__fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
25720     \int_value:w #1
25721   \else:
25722     0

```

```

25723         \msg_expandable_error:nnn
25724         { fp } { after-e } { dimension~#1 }
25725     \fi:
25726     \prg_return_false:
25727 \fi:
25728 \else:
25729     0
25730     \msg_expandable_error:nnn
25731     { fp } { missing } { exponent }
25732     \prg_return_true:
25733 \fi:
25734 }

```

(End of definition for __fp_parse_exponent_keep:NTF.)

76.5 Constants, functions and prefix operators

76.5.1 Prefix operators

__fp_parse_prefix_+:Nw A unary + does nothing: we should continue looking for a number.

```

25735 \cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw

```

(End of definition for __fp_parse_prefix_+:Nw.)

__fp_parse_apply_function:NNNwN Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is *e.g.*, __fp_sin_o:w, and expands once after the calculation, #4 is the operand, and #5 is a __fp_parse_infix...:N function. We feed the data #2, and the argument #4, to the function #3, which expands \exp:w thus the infix function #5.

```

25736 \cs_new:Npn \__fp_parse_apply_function:NNNwN #1#2#3#4#5
25737 {
25738     #3 #2 #4 @
25739     \exp:w \exp_end_continue_f:w #5 #1
25740 }

```

(End of definition for __fp_parse_apply_function:NNNwN.)

__fp_parse_apply_unary:NNNwN In contrast to __fp_parse_apply_function:NNNwN, this checks that the operand #4 is a single argument (namely there is a single __fp_sep:). We use the fact that any floating point starts with a “safe” token like \s__fp. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the `expl3` function that computes it, such as __fp_sin_o:w.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```

25741 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4#5
25742 {
25743     \__fp_parse_apply_unary_chk:NwNw #4 @ \__fp_sep: . \s__fp_stop
25744     \__fp_parse_apply_unary_type:NNN
25745     #3 #2 #4 @
25746     \exp:w \exp_end_continue_f:w #5 #1
25747 }
25748 \cs_new:Npn \__fp_parse_apply_unary_chk:NwNw #1#2 \__fp_sep: #3#4 \s__fp_stop

```

```

25749 {
25750   \if_meaning:w @ #3 \else:
25751     \token_if_eq_meaning:NNTF . #3
25752     { \__fp_parse_apply_unary_chk:nNNNNw { no } }
25753     { \__fp_parse_apply_unary_chk:nNNNNw { multi } }
25754   \fi:
25755 }
25756 \cs_new:Npn \__fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
25757 {
25758   #2
25759   \__fp_error:nffn { #1-arg } { \__fp_func_to_name:N #4 } { } { }
25760   \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
25761 }
25762 \cs_new:Npn \__fp_parse_apply_unary_type:NNN #1#2#3
25763 {
25764   \__fp_change_func_type:NNN #3 #1 \__fp_parse_apply_unary_error:NNw
25765   #2 #3
25766 }
25767 \cs_new:Npn \__fp_parse_apply_unary_error:NNw #1#2#3 @
25768 { \__fp_invalid_operation_o:fw { \__fp_func_to_name:N #1 } #3 }

```

(End of definition for __fp_parse_apply_unary:NNNwN and others.)

__fp_parse_prefix_-:Nw
 __fp_parse_prefix_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c__fp_prec_not_int of the unary operator, then call the appropriate __fp_⟨operation⟩_o:w function, where the ⟨operation⟩ is set_sign or not.

```

25769 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
25770 {
25771   \cs_new:cpn { \__fp_parse_prefix_ #1 :Nw } ##1
25772   {
25773     \exp_after:wN \__fp_parse_apply_unary:NNNwN
25774     \exp_after:wN ##1
25775     \exp_after:wN #4
25776     \exp_after:wN #3
25777     \exp:w
25778     \if_int_compare:w #2 < ##1
25779       \__fp_parse_operand:Nw ##1
25780     \else:
25781       \__fp_parse_operand:Nw #2
25782     \fi:
25783     \__fp_parse_expand:w
25784   }
25785 }
25786 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
25787 \__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?

```

(End of definition for __fp_parse_prefix_-:Nw and __fp_parse_prefix_!:Nw.)

__fp_parse_prefix_:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to __fp_parse_one_digit:NN but calls __fp_parse_strim_zeros:N to trim zeros after the decimal point, rather than the trim_zeros function for zeros before the decimal point.

```

25788 \cs_new:cpn { __fp_parse_prefix_:Nw } #1
25789 {
25790   \exp_after:wN \__fp_parse_infix_after_operand:NwN
25791   \exp_after:wN #1
25792   \exp:w \exp_end_continue_f:w
25793   \exp_after:wN \__fp_sanitizewN
25794   \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
25795 }

```

(End of definition for __fp_parse_prefix_:Nw.)

```

\__fp_parse_prefix_(:Nw
\__fp_parse_lparen_after:NwN

```

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is \c__fp_prec_func_int we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: \c__fp_prec_comma_int for the case of arguments, \c__fp_prec_tuple_int for the case of tuples. Once the operand is found, the lparen_after auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```

25796 \cs_new:cpn { __fp_parse_prefix_(:Nw } #1
25797 {
25798   \exp_after:wN \__fp_parse_lparen_after:NwN
25799   \exp_after:wN #1
25800   \exp:w
25801   \if_int_compare:w #1 = \c__fp_prec_func_int
25802     \__fp_parse_operand:Nw \c__fp_prec_comma_int
25803   \else:
25804     \__fp_parse_operand:Nw \c__fp_prec_tuple_int
25805   \fi:
25806   \__fp_parse_expand:w
25807 }
25808 \cs_new:Npe \__fp_parse_lparen_after:NwN #1#2 @ #3
25809 {
25810   \exp_not:N \token_if_eq_meaning:NNTF #3
25811   \exp_not:c { __fp_parse_infix_):N }
25812   {
25813     \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
25814     \exp_not:N \exp_after:wN
25815     \exp_not:N \__fp_parse_infix_after_paren:NN
25816     \exp_not:N \exp_after:wN #1
25817     \exp_not:N \exp:w
25818     \exp_not:N \__fp_parse_expand:w
25819   }
25820   {
25821     \exp_not:N \msg_expandable_error:nnn
25822     { fp } { missing } { ) }
25823     \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
25824     #2 @
25825     \exp_not:N \use_none:n #3
25826   }
25827 }

```

(End of definition for __fp_parse_prefix_(:Nw and __fp_parse_lparen_after:NwN.)

`\__fp_parse_prefix_):Nw` The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```

25828 \cs_new:cpn { __fp_parse_prefix_):Nw } #1
25829 {
25830   \if_int_compare:w #1 = \c__fp_prec_comma_int
25831   \else:
25832     \if_int_compare:w #1 = \c__fp_prec_tuple_int
25833     \exp_after:wN \c__fp_empty_tuple_fp \exp:w
25834   \else:
25835     \msg_expandable_error:nnn
25836       { fp } { missing-number } { ) }
25837     \exp_after:wN \c_nan_fp \exp:w
25838   \fi:
25839   \exp_end_continue_f:w
25840 \fi:
25841 \__fp_parse_infix_after_paren:NN #1 )
25842 }

```

(End of definition for __fp_parse_prefix_):Nw.)

76.5.2 Constants

`\__fp_parse_word_inf:N` Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```

\__fp_parse_word_nan:N
\__fp_parse_word_pi:N
\__fp_parse_word_deg:N
\__fp_parse_word_true:N
\__fp_parse_word_false:N
25843 \cs_set_protected:Npn \__fp_tmp:w #1 #2
25844 {
25845   \cs_new:cpn { __fp_parse_word_#1:N }
25846     { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
25847 }
25848 \__fp_tmp:w { inf } \c_inf_fp
25849 \__fp_tmp:w { nan } \c_nan_fp
25850 \__fp_tmp:w { pi } \c_pi_fp
25851 \__fp_tmp:w { deg } \c_one_degree_fp
25852 \__fp_tmp:w { true } \c_one_fp
25853 \__fp_tmp:w { false } \c_zero_fp

```

(End of definition for __fp_parse_word_inf:N and others.)

`\__fp_parse_caseless_inf:N` Copies of `\__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```

\__fp_parse_caseless_infinity:N
\__fp_parse_caseless_nan:N
25854 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
25855 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
25856 \cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N

```

(End of definition for __fp_parse_caseless_inf:N, __fp_parse_caseless_infinity:N, and __fp_parse_caseless_nan:N.)

`\__fp_parse_word_pt:N` Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

\__fp_parse_word_in:N
\__fp_parse_word_pc:N
\__fp_parse_word_cm:N
\__fp_parse_word_mm:N
\__fp_parse_word_dd:N
\__fp_parse_word_cc:N
\__fp_parse_word_nd:N
\__fp_parse_word_nc:N
\__fp_parse_word_bp:N
\__fp_parse_word_sp:N
25857 \cs_set_protected:Npn \__fp_tmp:w #1 #2
25858 {
25859   \cs_new:cpn { __fp_parse_word_#1:N }
25860   {

```

```

25861         \_fp_exp_after_f:nw { \_fp_parse_infix:NN }
25862         \s__fp \_fp_chk:w 10 #2 \_fp_sep:
25863     }
25864 }
25865 \_fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
25866 \_fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
25867 \_fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
25868 \_fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
25869 \_fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
25870 \_fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
25871 \_fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
25872 \_fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
25873 \_fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
25874 \_fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
25875 \_fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End of definition for _fp_parse_word_pt:N and others.)

_fp_parse_word_em:N The font-dependent units em and ex must be evaluated on the fly. We reuse an auxiliary of \dim_to_fp:n.

```

25876 \tl_map_inline:nn { {em} {ex} }
25877 {
25878     \cs_new:cpn { __fp_parse_word_#1:N }
25879     {
25880         \exp_after:wN \_fp_from_dim_test:ww
25881         \exp_after:wN 0 \exp_after:wN ,
25882         \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN \_fp_sep:
25883         \exp:w \exp_end_continue_f:w \_fp_parse_infix:NN
25884     }
25885 }

```

(End of definition for _fp_parse_word_em:N and _fp_parse_word_ex:N.)

76.5.3 Functions

```

\_fp_parse_unary_function:NNN
\_fp_parse_function:NNN
25886 \cs_new:Npn \_fp_parse_unary_function:NNN #1#2#3
25887 {
25888     \exp_after:wN \_fp_parse_apply_unary:NNNwN
25889     \exp_after:wN #3
25890     \exp_after:wN #2
25891     \exp_after:wN #1
25892     \exp:w
25893     \_fp_parse_operand:Nw \c__fp_prec_func_int \_fp_parse_expand:w
25894 }
25895 \cs_new:Npn \_fp_parse_function:NNN #1#2#3
25896 {
25897     \exp_after:wN \_fp_parse_apply_function:NNNwN
25898     \exp_after:wN #3
25899     \exp_after:wN #2
25900     \exp_after:wN #1
25901     \exp:w
25902     \_fp_parse_operand:Nw \c__fp_prec_func_int \_fp_parse_expand:w
25903 }

```


(End of definition for `\__fp_parse_unary_function:NNN` and `\__fp_parse_function:NNN`.)

76.6 Main functions

`\__fp_parse:n` Start an `\exp:w` expansion so that `\__fp_parse:n` expands in two steps. The `\__fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c__fp_prec_end_int` or less, namely, the end of the expression. The marker `\s__fp_expr_mark` indicates that the next token is an already parsed version of an infix operator, and `\__fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:.`

```

25904 \cs_new:Npn \__fp_parse:n #1
25905 {
25906   \exp:w
25907     \group_align_safe_begin:
25908     \exp_after:wN \group_align_safe_end:
25909     \exp_after:wN \__fp_parse_after:ww
25910     \exp:w
25911       \__fp_parse_operand:Nw \c__fp_prec_end_int
25912       \__fp_parse_expand:w #1
25913       \s__fp_expr_mark \__fp_parse_infix_end:N
25914       \s__fp_expr_stop
25915     \exp_end:
25916 }
25917 \cs_new:Npn \__fp_parse_after:ww
25918   #1@ \__fp_parse_infix_end:N \s__fp_expr_stop #2 { #2 #1 }
25919 \cs_new:Npn \__fp_parse_o:n #1
25920 {
25921   \exp:w
25922     \group_align_safe_begin:
25923     \exp_after:wN \group_align_safe_end:
25924     \exp_after:wN \__fp_parse_after:ww
25925     \exp:w
25926       \__fp_parse_operand:Nw \c__fp_prec_end_int
25927       \__fp_parse_expand:w #1
25928       \s__fp_expr_mark \__fp_parse_infix_end:N
25929     \s__fp_expr_stop
25930   {
25931     \exp_end_continue_f:w
25932     \__fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
25933   }
25934 }
```

(End of definition for `\__fp_parse:n`, `\__fp_parse_o:n`, and `\__fp_parse_after:ww`.)

`\__fp_parse_operand:Nw` This is just a shorthand which sets up both `\__fp_parse_continue:NwN` and `\__fp_parse_one:Nw` with the same precedence. Note the trailing `\exp:w`.

```

25935 \cs_new:Npn \__fp_parse_operand:Nw #1
25936 {
25937   \exp_end_continue_f:w
25938   \exp_after:wN \__fp_parse_continue:NwN
25939   \exp_after:wN #1
```

```

25940 \exp:w \exp_end_continue_f:w
25941 \exp_after:wN \_fp_parse_one:Nw
25942 \exp_after:wN #1
25943 \exp:w
25944 }
25945 \cs_new:Npn \_fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End of definition for _fp_parse_operand:Nw and _fp_parse_continue:NwN.)

_fp_parse_apply_binary:NwNwN
_fp_parse_apply_binary_chk:NN
_fp_parse_apply_binary_error:NNN

Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

25946 \cs_new:Npn \_fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
25947 {
25948 \exp_after:wN \_fp_parse_continue:NwN
25949 \exp_after:wN #1
25950 \exp:w \exp_end_continue_f:w
25951 \exp_after:wN \_fp_parse_apply_binary_chk:NN
25952 \cs:w
25953 \_fp
25954 \_fp_type_from_scan:N #2
25955 _#4
25956 \_fp_type_from_scan:N #5
25957 _o:ww
25958 \cs_end:
25959 #4
25960 #2#3 #5#6
25961 \exp:w \exp_end_continue_f:w #7 #1
25962 }
25963 \cs_new:Npn \_fp_parse_apply_binary_chk:NN #1#2
25964 {
25965 \if_meaning:w \scan_stop: #1
25966 \_fp_parse_apply_binary_error:NNN #2
25967 \fi:
25968 #1
25969 }
25970 \cs_new:Npn \_fp_parse_apply_binary_error:NNN #1#2#3
25971 {
25972 #2
25973 \_fp_invalid_operation_o:Nww #1
25974 }

```

(End of definition for _fp_parse_apply_binary:NwNwN, _fp_parse_apply_binary_chk:NN, and _fp_parse_apply_binary_error:NNN.)

_fp_binary_type_o:Nww
_fp_binary_rev_type_o:Nww

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The rev version swaps its arguments before doing this.

```

25975 \cs_new:Npn \_fp_binary_type_o:Nww #1 #2#3 \_fp_sep: #4
25976 {
25977 \exp_after:wN \_fp_parse_apply_binary_chk:NN
25978 \cs:w
25979 \_fp
25980 \_fp_type_from_scan:N #2

```

```

25981     _ #1
25982     \_fp_type_from_scan:N #4
25983     _o:ww
25984     \cs_end:
25985     #1
25986     #2 #3 \_fp_sep: #4
25987   }
25988 \cs_new:Npn \_fp_binary_rev_type_o:Nww #1 #2#3 \_fp_sep: #4#5 \_fp_sep:
25989 {
25990   \exp_after:wN \_fp_parse_apply_binary_chk:NN
25991   \cs:w
25992     _fp
25993     \_fp_type_from_scan:N #4
25994     _ #1
25995     \_fp_type_from_scan:N #2
25996     _o:ww
25997     \cs_end:
25998     #1
25999     #4 #5 \_fp_sep: #2 #3 \_fp_sep:
26000 }

```

(End of definition for _fp_binary_type_o:Nww and _fp_binary_rev_type_o:Nww.)

76.7 Infix operators

_fp_parse_infix_after_operand:NwN

```

26001 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2\_fp_sep:
26002 {
26003   \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
26004   #2\_fp_sep:
26005 }
26006 \cs_new:Npn \_fp_parse_infix:NN #1 #2
26007 {
26008   \if_catcode:w \scan_stop: \exp_not:N #2
26009   \if:w 0 \_fp_str_if_eq:nn { \s_fp_expr_mark } { \exp_not:N #2 }
26010   \exp_after:wN \exp_after:wN
26011   \exp_after:wN \_fp_parse_infix_mark:NNN
26012   \else:
26013     \exp_after:wN \exp_after:wN
26014     \exp_after:wN \_fp_parse_infix_juxt:N
26015   \fi:
26016   \else:
26017     \if_int_compare:w
26018       \_fp_int_eval:w
26019       ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
26020       = 3 \exp_stop_f:
26021     \exp_after:wN \exp_after:wN
26022     \exp_after:wN \_fp_parse_infix_juxt:N
26023   \else:
26024     \exp_after:wN \_fp_parse_infix_check:NNN
26025   \cs:w
26026     \_fp_parse_infix_ \token_to_str:N #2 :N
26027     \exp_after:wN \exp_after:wN \exp_after:wN

```

```

26028         \cs_end:
26029     \fi:
26030 \fi:
26031     #1
26032     #2
26033 }
26034 \cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
26035 {
26036     \if_meaning:w \scan_stop: #1
26037     \msg_expandable_error:nnn
26038     { fp } { missing } { * }
26039     \exp_after:wN \__fp_parse_infix_mul:N
26040     \exp_after:wN #2
26041     \exp_after:wN #3
26042 \else:
26043     \exp_after:wN #1
26044     \exp_after:wN #2
26045     \exp:w \exp_after:wN \__fp_parse_expand:w
26046 \fi:
26047 }

```

(End of definition for __fp_parse_infix_after_operand:NwN.)

__fp_parse_infix_after_paren:NN Variant of __fp_parse_infix:NN for use after a closing parenthesis. The only difference is that __fp_parse_infix_juxt:N is replaced by __fp_parse_infix_mul:N.

```

26048 \cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
26049 {
26050     \if_catcode:w \scan_stop: \exp_not:N #2
26051     \if:w 0 \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
26052     \exp_after:wN \exp_after:wN
26053     \exp_after:wN \__fp_parse_infix_mark:NNN
26054 \else:
26055     \exp_after:wN \exp_after:wN
26056     \exp_after:wN \__fp_parse_infix_mul:N
26057 \fi:
26058 \else:
26059     \if_int_compare:w
26060     \__fp_int_eval:w
26061     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
26062     = 3 \exp_stop_f:
26063     \exp_after:wN \exp_after:wN
26064     \exp_after:wN \__fp_parse_infix_mul:N
26065 \else:
26066     \exp_after:wN \__fp_parse_infix_check:NNN
26067     \cs:w
26068     __fp_parse_infix_ \token_to_str:N #2 :N
26069     \exp_after:wN \exp_after:wN \exp_after:wN
26070     \cs_end:
26071 \fi:
26072 \fi:
26073     #1
26074     #2
26075 }

```

(End of definition for __fp_parse_infix_after_paren:NN.)

76.7.1 Closing parentheses and commas

`\__fp_parse_infix_mark:NNN` As an infix operator, `\s__fp_expr_mark` means that the next token (#3) has already gone through `\__fp_parse_infix:NN` and should be provided the precedence #1. The scan mark #2 is discarded.

```
26076 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

(End of definition for __fp_parse_infix_mark:NNN.)

`\__fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
26077 \cs_new:Npn \__fp_parse_infix_end:N #1
26078 { @ \use_none:n \__fp_parse_infix_end:N }
```

(End of definition for __fp_parse_infix_end:N.)

`\__fp_parse_infix_):N` This is very similar to `\__fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence `\c__fp_prec_end_int`.

```
26079 \cs_set_protected:Npn \__fp_tmp:w #1
26080 {
26081   \cs_new:Npn #1 ##1
26082   {
26083     \if_int_compare:w ##1 > \c__fp_prec_end_int
26084       \exp_after:wN @
26085       \exp_after:wN \use_none:n
26086       \exp_after:wN #1
26087     \else:
26088       \msg_expandable_error:nnn { fp } { extra } { } }
26089     \exp_after:wN \__fp_parse_infix:NN
26090     \exp_after:wN ##1
26091     \exp:w \exp_after:wN \__fp_parse_expand:w
26092   \fi:
26093 }
26094 }
26095 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_):N }
```

(End of definition for __fp_parse_infix_):N.)

`\__fp_parse_infix_,:N` As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call `\__fp_parse_operand:Nw` to read more comma-delimited arguments that `\__fp_parse_infix_comma:w` simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call `\__fp_parse_apply_comma:NwNwN` whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to `\__fp_parse_apply_binary:NwNwN` this function's operands are not single-object arrays.

```
26096 \cs_set_protected:Npn \__fp_tmp:w #1
26097 {
26098   \cs_new:Npn #1 ##1
26099   {
26100     \if_int_compare:w ##1 > \c__fp_prec_comma_int
26101       \exp_after:wN @
26102       \exp_after:wN \use_none:n
26103       \exp_after:wN #1
```

```

26104         \else:
26105             \if_int_compare:w ##1 < \c__fp_prec_comma_int
26106                 \exp_after:wN @
26107                 \exp_after:wN \__fp_parse_apply_comma:NwNwN
26108                 \exp_after:wN ,
26109                 \exp:w
26110             \else:
26111                 \exp_after:wN \__fp_parse_infix_comma:w
26112                 \exp:w
26113             \fi:
26114             \__fp_parse_operand:Nw \c__fp_prec_comma_int
26115             \exp_after:wN \__fp_parse_expand:w
26116         \fi:
26117     }
26118 }
26119 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_,:N }
26120 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
26121 { #1 @ \use_none:n }
26122 \cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
26123 {
26124     \exp_after:wN \__fp_parse_continue:NwN
26125     \exp_after:wN #1
26126     \exp:w \exp_end_continue_f:w
26127     \__fp_exp_after_tuple_f:nw { }
26128     \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } \__fp_sep:
26129     #5 #1
26130 }

```

(End of definition for __fp_parse_infix_,:N, __fp_parse_infix_comma:w, and __fp_parse_apply_comma:NwNwN.)

76.7.2 Usual infix operators

As described in the “work plan”, each infix operator has an associated \..._infix... function, a computing function, and precedence, given as arguments to __fp_tmp:w. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

\__fp_parse_infix_+:N
\__fp_parse_infix_-:N
\__fp_parse_infix_juxt:N
\__fp_parse_infix_/:N
\__fp_parse_infix_mul:N
\__fp_parse_infix_and:N
\__fp_parse_infix_or:N
\__fp_parse_infix_^:N
26131 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
26132 {
26133     \cs_new:Npn #1 ##1
26134     {
26135         \if_int_compare:w ##1 < #3
26136             \exp_after:wN @
26137             \exp_after:wN \__fp_parse_apply_binary:NwNwN
26138             \exp_after:wN #2
26139             \exp:w
26140             \__fp_parse_operand:Nw #4
26141             \exp_after:wN \__fp_parse_expand:w
26142         \else:
26143             \exp_after:wN @
26144             \exp_after:wN \use_none:n
26145             \exp_after:wN #1
26146         \fi:
26147     }

```

```

26148 }
26149 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_~:N } ^
26150 \c__fp_prec_hatii_int \c__fp_prec_hat_int
26151 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_juxt:N } *
26152 \c__fp_prec_juxt_int \c__fp_prec_juxt_int
26153 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:N } /
26154 \c__fp_prec_times_int \c__fp_prec_times_int
26155 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
26156 \c__fp_prec_times_int \c__fp_prec_times_int
26157 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix -:N } -
26158 \c__fp_prec_plus_int \c__fp_prec_plus_int
26159 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_+:N } +
26160 \c__fp_prec_plus_int \c__fp_prec_plus_int
26161 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
26162 \c__fp_prec_and_int \c__fp_prec_and_int
26163 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N } |
26164 \c__fp_prec_or_int \c__fp_prec_or_int

```

(End of definition for `\__fp_parse_infix_+:N` and others.)

76.7.3 Juxtaposition

`\__fp_parse_infix_(:N` When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using `\__fp_parse_infix_mul:N`.

```

26165 \cs_new:cpn { __fp_parse_infix_(:N } #1
26166 { \__fp_parse_infix_mul:N #1 ( }

```

(End of definition for `\__fp_parse_infix_(:N`.)

76.7.4 Multi-character cases

`\__fp_parse_infix_*:N`

```

26167 \cs_set_protected:Npn \__fp_tmp:w #1
26168 {
26169   \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
26170   {
26171     \if:w * \exp_not:N ##2
26172       \exp_after:wN #1
26173       \exp_after:wN ##1
26174     \else:
26175       \exp_after:wN \__fp_parse_infix_mul:N
26176       \exp_after:wN ##1
26177       \exp_after:wN ##2
26178     \fi:
26179   }
26180 }
26181 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_~:N }

```

(End of definition for `\__fp_parse_infix_*:N`.)

`\__fp_parse_infix_|:Nw`

`\__fp_parse_infix_&:Nw`

```

26182 \cs_set_protected:Npn \__fp_tmp:w #1#2#3
26183 {

```

```

26184 \cs_new:Npn #1 ##1##2
26185 {
26186   \if:w #2 \exp_not:N ##2
26187     \exp_after:wN #1
26188     \exp_after:wN ##1
26189     \exp:w \exp_after:wN \__fp_parse_expand:w
26190   \else:
26191     \exp_after:wN #3
26192     \exp_after:wN ##1
26193     \exp_after:wN ##2
26194   \fi:
26195 }
26196 }
26197 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_|:N } | \__fp_parse_infix_or:N
26198 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_&:N } & \__fp_parse_infix_and:N

```

(End of definition for __fp_parse_infix_|:Nw and __fp_parse_infix_&:Nw.)

76.7.5 Ternary operator

```

\__fp_parse_infix_?:N
\__fp_parse_infix_::N
26199 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
26200 {
26201   \cs_new:Npn #1 ##1
26202   {
26203     \if_int_compare:w ##1 < \c__fp_prec_quest_int
26204       #4
26205       \exp_after:wN @
26206       \exp_after:wN #2
26207       \exp:w
26208       \__fp_parse_operand:Nw #3
26209       \exp_after:wN \__fp_parse_expand:w
26210     \else:
26211       \exp_after:wN @
26212       \exp_after:wN \use_none:n
26213       \exp_after:wN #1
26214     \fi:
26215   }
26216 }
26217 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
26218 \__fp_ternary:NwwN \c__fp_prec_quest_int { }
26219 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_::N }
26220 \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
26221 {
26222   \msg_expandable_error:nnnn
26223   { fp } { missing } { ? } { ~for~?: }
26224 }

```

(End of definition for __fp_parse_infix_?:N and __fp_parse_infix_::N.)

76.7.6 Comparisons

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_excl_error:
\__fp_parse_compare:NNNNNNN
\__fp_parse_compare_auxi:NNNNNN
\__fp_parse_compare_auxii:NNNNN
\__fp_parse_compare_end:NNNw
\__fp_compare:wNNNNw
26225 \cs_new:cpn { __fp_parse_infix_<:N } #1

```



```

26226 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
26227 \cs_new:cpn { \__fp_parse_infix_=:N } #1
26228 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
26229 \cs_new:cpn { \__fp_parse_infix_>:N } #1
26230 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
26231 \cs_new:cpn { \__fp_parse_infix_!:N } #1
26232 {
26233   \exp_after:wN \__fp_parse_compare:NNNNNNN
26234   \exp_after:wN #1
26235   \exp_after:wN 0
26236   \exp_after:wN 1
26237   \exp_after:wN 1
26238   \exp_after:wN 1
26239   \exp_after:wN 1
26240 }
26241 \cs_new:Npn \__fp_parse_excl_error:
26242 {
26243   \msg_expandable_error:nnnn
26244   { fp } { missing } { = } { ~after~!. }
26245 }
26246 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
26247 {
26248   \if_int_compare:w #1 < \c__fp_prec_comp_int
26249     \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
26250     \exp_after:wN \__fp_parse_excl_error:
26251   \else:
26252     \exp_after:wN @
26253     \exp_after:wN \use_none:n
26254     \exp_after:wN \__fp_parse_compare:NNNNNNN
26255   \fi:
26256 }
26257 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
26258 {
26259   \if_case:w
26260     \__fp_int_eval:w \exp_after:wN ' \token_to_str:N #7 - ' <
26261     \__fp_int_eval_end:
26262     \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
26263   \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
26264   \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
26265   \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
26266   \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
26267   \fi:
26268 }
26269 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
26270 {
26271   \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
26272   \exp_after:wN \prg_do_nothing:
26273   \exp_after:wN #1
26274   \exp_after:wN #2
26275   \exp_after:wN #3
26276   \exp_after:wN #4
26277   \exp_after:wN #5
26278   \exp:w \exp_after:wN \__fp_parse_expand:w
26279 }

```

```

26280 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
26281 {
26282   \fi:
26283   \exp_after:wN @
26284   \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
26285   \exp_after:wN \c_one_fp
26286   \exp_after:wN #1
26287   \exp_after:wN #2
26288   \exp_after:wN #3
26289   \exp_after:wN #4
26290   \exp:w
26291   \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
26292 }
26293 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
26294   #1 #2@ #3 #4#5#6#7 #8@ #9
26295 {
26296   \if_int_odd:w
26297     \if_meaning:w \c_zero_fp #3
26298     0
26299     \else:
26300       \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
26301       #5 \or: #6 \or: #7 \else: #4
26302       \fi:
26303       \fi:
26304       \exp_stop_f:
26305       \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
26306       \exp_after:wN \c_one_fp
26307     \else:
26308       \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
26309       \exp_after:wN \c_zero_fp
26310     \fi:
26311     #1 #8 #9
26312 }
26313 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3\__fp_sep: #4
26314 {
26315   \if_meaning:w \__fp_parse_compare:NNNNNNN #4
26316     \exp_after:wN \__fp_parse_continue_compare:NNwNN
26317     \exp_after:wN #1
26318     \exp_after:wN #2
26319     \exp:w \exp_end_continue_f:w
26320     \__fp_exp_after_o:w #3\__fp_sep:
26321     \exp:w \exp_end_continue_f:w
26322   \else:
26323     \exp_after:wN \__fp_parse_continue:NwN
26324     \exp_after:wN #2
26325     \exp:w \exp_end_continue_f:w
26326     \exp_after:wN #1
26327     \exp:w \exp_end_continue_f:w
26328   \fi:
26329   #4 #2
26330 }
26331 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
26332 { #4 #2 #3@ #1 }

```

(End of definition for __fp_parse_infix_<:N and others.)

76.8 Tools for functions

`\_fp_parse_function_all_fp_o:fnw` Followed by `{⟨function name⟩} {⟨code⟩} ⟨float array⟩ @` this checks all floats are floating point numbers (no tuples).

```

26333 \cs_new:Npn \_fp_parse_function_all_fp_o:fnw #1#2#3 @
26334 {
26335   \_fp_array_if_all_fp:nTF {#3}
26336   { #2 #3 @ }
26337   {
26338     \_fp_error:nffn { bad-args }
26339     {#1}
26340     { \fp_to_tl:n { \s__fp_tuple \_fp_tuple_chk:w {#3} \_fp_sep: } }
26341     { }
26342     \exp_after:wN \c_nan_fp
26343   }
26344 }

```

(End of definition for `\_fp_parse_function_all_fp_o:fnw`.)

`\_fp_parse_function_one_two:nnw` This is followed by `{⟨function name⟩} {⟨code⟩} ⟨float array⟩ @`. It checks that the `⟨float array⟩` consists of one or two floating point numbers (not tuples), then leaves `\_fp_parse_function_one_two_error_o:w` the `⟨code⟩` (if there is one float) or its tail (if there are two floats) followed by the `⟨float array⟩`. The `⟨code⟩` should start with a single token such as `\_fp_atan_default:w` that deals with the single-float case.

The first `\_fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```

26345 \cs_new:Npn \_fp_parse_function_one_two:nnw #1#2#3
26346 {
26347   \_fp_if_type_fp:NTwFw
26348   #3 { } \s__fp \_fp_parse_function_one_two_error_o:w \s__fp_stop
26349   \_fp_parse_function_one_two_aux:nnw {#1} {#2} #3
26350 }
26351 \cs_new:Npn \_fp_parse_function_one_two_error_o:w #1#2#3#4 @
26352 {
26353   \_fp_error:nffn { bad-args }
26354   {#2}
26355   { \fp_to_tl:n { \s__fp_tuple \_fp_tuple_chk:w {#4} \_fp_sep: } }
26356   { }
26357   \exp_after:wN \c_nan_fp
26358 }
26359 \cs_new:Npn \_fp_parse_function_one_two_aux:nnw #1#2 #3\_fp_sep: #4
26360 {
26361   \_fp_if_type_fp:NTwFw
26362   #4 { }
26363   \s__fp
26364   {
26365     \if_meaning:w @ #4
26366     \exp_after:wN \use_iv:nnnn
26367     \fi:
26368     \_fp_parse_function_one_two_error_o:w
26369   }

```

```

26370     \s__fp_stop
26371     \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3\__fp_sep: #4
26372 }
26373 \cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3\__fp_sep: #4\__fp_sep: #5
26374 {
26375     \if_meaning:w @ #5 \else:
26376         \exp_after:wN \__fp_parse_function_one_two_error_o:w
26377     \fi:
26378     \use_ii:nn {#1} { \use_none:n #2 } #3\__fp_sep: #4\__fp_sep: #5
26379 }

```

(End of definition for __fp_parse_function_one_two:nnw and others.)

__fp_tuple_map_o:nw Apply #1 to all items in the following tuple and expand once afterwards. The code #1
 __fp_tuple_map_loop_o:nw should itself expand once after its result.

```

26380 \cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 \__fp_sep:
26381 {
26382     \exp_after:wN \s__fp_tuple
26383     \exp_after:wN \__fp_tuple_chk:w
26384     \exp_after:wN {
26385         \exp:w \exp_end_continue_f:w
26386         \__fp_tuple_map_loop_o:nw {#1} #2
26387         { \s__fp \prg_break: } \__fp_sep:
26388         \prg_break_point:
26389     \exp_after:wN } \exp_after:wN \__fp_sep:
26390 }
26391 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 \__fp_sep:
26392 {
26393     \use_none:n #2
26394     #1 #2 #3 \__fp_sep:
26395     \exp:w \exp_end_continue_f:w
26396     \__fp_tuple_map_loop_o:nw {#1}
26397 }

```

(End of definition for __fp_tuple_map_o:nw and __fp_tuple_map_loop_o:nw.)

__fp_tuple_mapthread_o:nnw Apply #1 to pairs of items in the two following tuples and expand once afterwards.
 __fp_tuple_mapthread_loop_o:nw

```

26398 \cs_new:Npn \__fp_tuple_mapthread_o:nnw #1
26399     \s__fp_tuple \__fp_tuple_chk:w #2 \__fp_sep:
26400     \s__fp_tuple \__fp_tuple_chk:w #3 \__fp_sep:
26401 {
26402     \exp_after:wN \s__fp_tuple
26403     \exp_after:wN \__fp_tuple_chk:w
26404     \exp_after:wN {
26405         \exp:w \exp_end_continue_f:w
26406         \__fp_tuple_mapthread_loop_o:nw {#1}
26407         #2 { \s__fp \prg_break: } \__fp_sep: @
26408         #3 { \s__fp \prg_break: } \__fp_sep:
26409         \prg_break_point:
26410     \exp_after:wN } \exp_after:wN \__fp_sep:
26411 }
26412 \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 \__fp_sep: #4 @ #5#6 \__fp_sep:
26413 {
26414     \use_none:n #2

```

```

26415     \use_none:n #5
26416     #1 #2 #3 \__fp_sep: #5 #6 \__fp_sep:
26417     \exp:w \exp_end_continue_f:w
26418     \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
26419 }

```

(End of definition for __fp_tuple_mapthread_o:nww and __fp_tuple_mapthread_loop_o:nw.)

76.9 Messages

```

26420 \msg_new:nnn { fp } { deprecated }
26421 { '#1'~deprecated;~use~'#2' }
26422 \msg_new:nnn { fp } { unknown-fp-word }
26423 { Unknown~fp~word~#1. }
26424 \msg_new:nnn { fp } { missing }
26425 { Missing~#1~inserted #2. }
26426 \msg_new:nnn { fp } { extra }
26427 { Extra~#1~ignored. }
26428 \msg_new:nnn { fp } { early-end }
26429 { Premature~end~in~fp~expression. }
26430 \msg_new:nnn { fp } { after-e }
26431 { Cannot~use~#1 after~'e'. }
26432 \msg_new:nnn { fp } { missing-number }
26433 { Missing~number~before~'#1'. }
26434 \msg_new:nnn { fp } { unknown-symbol }
26435 { Unknown~symbol~#1~ignored. }
26436 \msg_new:nnn { fp } { extra-comma }
26437 { Unexpected~comma~turned~to~nan~result. }
26438 \msg_new:nnn { fp } { no-arg }
26439 { #1~got~no~argument;~used~nan. }
26440 \msg_new:nnn { fp } { multi-arg }
26441 { #1~got~more~than~one~argument;~used~nan. }
26442 \msg_new:nnn { fp } { bad-args }
26443 { Arguments~in~#1#2~are~invalid. }
26444 \msg_new:nnn { fp } { infity-pi }
26445 { Math~command~#1 is~not~an~fp }
26446 \cs_if_exist:cT { @unexpandable@protect }
26447 {
26448     \msg_new:nnn { fp } { robust-cmd }
26449     { Robust~command~#1 invalid~in~fp~expression! }
26450 }
26451 </code>

```

Chapter 77

l3fp-assign implementation

```
26452 <*code>
26453 <@@=fp>
```

77.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```
26454 \cs_new_protected:Npn \fp_new:N #1
26455   { \cs_new_eq:NN #1 \c_zero_fp }
26456 \cs_generate_variant:Nn \fp_new:N {c}
```

(End of definition for \fp_new:N. This function is documented on page 267.)

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

```
\fp_set:cn 26457 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_set:NV 26458   { \__kernel_tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_set:cV 26459 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_gset:Nn 26460   { \__kernel_tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 26461 \cs_new_protected:Npn \fp_const:Nn #1#2
\fp_gset:NV 26462   { \tl_const:Ne #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cV 26463 \cs_generate_variant:Nn \fp_set:Nn { NV , c , cV }
\fp_const:Nn 26464 \cs_generate_variant:Nn \fp_gset:Nn { NV , c , cV }
\fp_const:cn 26465 \cs_generate_variant:Nn \fp_const:Nn {c}
```

(End of definition for \fp_set:Nn, \fp_gset:Nn, and \fp_const:Nn. These functions are documented on page 267.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```
\fp_set_eq:cN 26466 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 26467 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 26468 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 26469 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc
```

(End of definition for \fp_set_eq:NN and \fp_gset_eq:NN. These functions are documented on page 267.)

```

\fp_zero:N Setting a floating point to zero: copy \c_zero_fp.
\fp_zero:c 26470 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 26471 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 26472 \cs_generate_variant:Nn \fp_zero:N { c }
26473 \cs_generate_variant:Nn \fp_gzero:N { c }

(End of definition for \fp_zero:N and \fp_gzero:N. These functions are documented on page 267.)

```

```

\fp_zero_new:N Set the floating point to zero, or define it if needed.
\fp_zero_new:c 26474 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 26475 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 26476 \cs_new_protected:Npn \fp_gzero_new:N #1
26477 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
26478 \cs_generate_variant:Nn \fp_zero_new:N { c }
26479 \cs_generate_variant:Nn \fp_gzero_new:N { c }

(End of definition for \fp_zero_new:N and \fp_gzero_new:N. These functions are documented on page 267.)

```

77.2 Updating values

These match the equivalent functions in l3int and l3skip.

```

\fp_add:Nn For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
\fp_add:cn is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
\fp_gadd:Nn the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
\fp_gadd:cn parentheses. As an optimization we use __fp_parse:n rather than \fp_eval:n, which
\fp_sub:Nn would convert the result away from the internal representation and back.
\fp_sub:cn 26480 \cs_new_protected:Npn \fp_add:Nn { __fp_add:NNNn \fp_set:Nn + }
\fp_gsub:Nn 26481 \cs_new_protected:Npn \fp_gadd:Nn { __fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cn 26482 \cs_new_protected:Npn \fp_sub:Nn { __fp_add:NNNn \fp_set:Nn - }
__fp_add:NNNn 26483 \cs_new_protected:Npn \fp_gsub:Nn { __fp_add:NNNn \fp_gset:Nn - }
26484 \cs_new_protected:Npn __fp_add:NNNn #1#2#3#4
26485 { #1 #3 { #3 #2 __fp_parse:n {#4} } }
26486 \cs_generate_variant:Nn \fp_add:Nn { c }
26487 \cs_generate_variant:Nn \fp_gadd:Nn { c }
26488 \cs_generate_variant:Nn \fp_sub:Nn { c }
26489 \cs_generate_variant:Nn \fp_gsub:Nn { c }

(End of definition for \fp_add:Nn and others. These functions are documented on page 267.)

```

77.3 Showing values

```

\fp_show:N This shows the result of computing its argument by passing the right data to \tl_show:n
\fp_show:c or \tl_log:n.
\fp_log:N 26490 \cs_new_protected:Npn \fp_show:N { __fp_show:NN \tl_show:n }
\fp_log:c 26491 \cs_generate_variant:Nn \fp_show:N { c }
__fp_show:NN 26492 \cs_new_protected:Npn \fp_log:N { __fp_show:NN \tl_log:n }
26493 \cs_generate_variant:Nn \fp_log:N { c }
26494 \cs_new_protected:Npn __fp_show:NN #1#2
26495 {
26496   __kernel_chk_tl_type:NnnT #2 { fp }

```

```

26497     { \exp_args:No \__fp_show_validate:n #2 }
26498     { \exp_args:Ne #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
26499   }

```

(End of definition for \fp_show:N, \fp_log:N, and __fp_show:NN. These functions are documented on page 278.)

```

\__fp_show_validate:n
\__fp_show_validate_aux:n
\__fp_show_validate:nn
\__fp_show_validate:w
\__fp_tuple_show_validate:w
\__fp_symbolic_show_validate:w

To support symbolic expression, validation has to be done recursively. Two \@@_show_validate:nn
wrappers are used to distinguish between initial and recursive calls, in which the former
provides a demo of possible forms a fp variable would have.

26500 \cs_new:Npn \__fp_show_validate:n #1
26501 {
26502   \__fp_show_validate:nn { #1 }
26503   {
26504     \s__fp \__fp_chk:w ??? \__fp_sep: or \iow_newline:
26505     \s__fp_tuple \__fp_tuple_chk:w ? \__fp_sep: or \iow_newline:
26506     \s__fp_symbolic \__fp_symbolic_chk:w ? , ? \__fp_sep:
26507   }
26508 }
26509 \cs_new:Npn \__fp_show_validate_aux:n #1
26510 {
26511   \__fp_show_validate:nn { #1 } { }
26512 }
26513 \cs_new:Npn \__fp_show_validate:nn #1#2
26514 {
26515   \tl_if_empty:nF { #1 }
26516   {
26517     \str_case:enF { \tl_head:n { #1 } }
26518     {
26519       { \s__fp }
26520       {
26521         \__fp_show_validate:w #1 \s__fp
26522         \__fp_chk:w ??? \__fp_sep: \s__fp_stop
26523       }
26524       { \s__fp_tuple }
26525       {
26526         \__fp_tuple_show_validate:w #1
26527         \s__fp_tuple \__fp_tuple_chk:w ?? \__fp_sep: \s__fp_stop
26528       }
26529       { \s__fp_symbolic }
26530       {
26531         \__fp_symbolic_show_validate:w #1
26532         \s__fp_symbolic \__fp_symbolic_chk:w ? , ?? \__fp_sep: \s__fp_stop
26533       }
26534     }
26535     { #2 }
26536   }
26537 }
26538 \cs_new:Npn \__fp_show_validate:w
26539 #1 \s__fp \__fp_chk:w #2#3#4#5 \__fp_sep: #6 \s__fp_stop
26540 {
26541   \str_if_eq:nnF { #2 } {?}
26542   {
26543     \token_if_eq_meaning:NNTF #2 1

```



```

26544         { \s__fp \__fp_chk:w #2 #3 { #4 } #5 \__fp_sep: }
26545         { \s__fp \__fp_chk:w #2 #3 #4 #5 \__fp_sep: }
26546         \__fp_show_validate_aux:n { #6 }
26547     }
26548 }
26549 \cs_new:Npn \__fp_tuple_show_validate:w
26550     #1 \s__fp_tuple \__fp_tuple_chk:w #2#3 \__fp_sep: #4 \s__fp_stop
26551 {
26552     \str_if_eq:nnF { #2 } {?}
26553     {
26554         \s__fp_tuple \__fp_tuple_chk:w { \__fp_show_validate_aux:n { #2 } }
26555         \__fp_sep:
26556     }
26557 }
26558 \cs_new:Npn \__fp_symbolic_show_validate:w
26559     #1 \s__fp_symbolic \__fp_symbolic_chk:w #2 , #3#4 \__fp_sep: #5 \s__fp_stop
26560 {
26561     \str_if_eq:nnF { #2 } {?}
26562     {
26563         \s__fp_symbolic \__fp_symbolic_chk:w \exp_not:n { #2 } ,
26564         { \__fp_show_validate_aux:n { #3 } }\__fp_sep:
26565         \__fp_show_validate_aux:n { #5 }
26566     }
26567 }

```

(End of definition for __fp_show_validate:n and others.)

\fp_show:n Use general tools.

\fp_log:n

```

26568 \cs_new_protected:Npn \fp_show:n
26569     { \__kernel_msg_show_eval:Nn \fp_to_tl:n }
26570 \cs_new_protected:Npn \fp_log:n
26571     { \__kernel_msg_log_eval:Nn \fp_to_tl:n }

```

(End of definition for \fp_show:n and \fp_log:n. These functions are documented on page 278.)

77.4 Some useful constants and scratch variables

\c_one_fp Some constants.

\c_e_fp

```

26572 \fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
26573 \fp_const:Nn \c_one_fp        { 1 }

```

(End of definition for \c_one_fp and \c_e_fp. These variables are documented on page 276.)

\c_pi_fp We simply round π to and $\pi/180$ to 16 significant digits.

\c_one_degree_fp

```

26574 \fp_const:Nn \c_pi_fp          { 3.141 5926 5358 9793 }
26575 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }

```

(End of definition for \c_pi_fp and \c_one_degree_fp. These variables are documented on page 276.)

\l_tmpa_fp Scratch variables are simply initialized there.

\l_tmpb_fp

```

26576 \fp_new:N \l_tmpa_fp
26577 \fp_new:N \l_tmpb_fp
26578 \fp_new:N \g_tmpa_fp
26579 \fp_new:N \g_tmpb_fp

```

(End of definition for `\l_tmpa_fp` and others. These variables are documented on page 276.)

26580 `</code>`

Chapter 78

l3fp-logic implementation

26581 $\langle *code \rangle$

26582 $\langle @@=fp \rangle$

$\backslash\_fp\_parse\_word\_max:N$

$\backslash\_fp\_parse\_word\_min:N$

Those functions may receive a variable number of arguments.

26583 $\backslash cs\_new:Npn \backslash\_fp\_parse\_word\_max:N$

26584 $\{ \backslash\_fp\_parse\_function:NNN \backslash\_fp\_minmax\_o:Nw 2 \}$

26585 $\backslash cs\_new:Npn \backslash\_fp\_parse\_word\_min:N$

26586 $\{ \backslash\_fp\_parse\_function:NNN \backslash\_fp\_minmax\_o:Nw 0 \}$

(End of definition for $\backslash\_fp\_parse\_word\_max:N$ and $\backslash\_fp\_parse\_word\_min:N$.)

78.1 Syntax of internal functions

- $\backslash\_fp\_compare\_npos:nwnw \{ \langle expo_1 \rangle \} \langle body_1 \rangle \backslash\_fp\_sep: \{ \langle expo_2 \rangle \} \langle body_2 \rangle \backslash\_fp\_sep:$
- $\backslash\_fp\_minmax\_o:Nw \langle sign \rangle \langle floating\ point\ array \rangle$
- $\backslash\_fp\_not\_o:w ? \langle floating\ point\ array \rangle$ (with one floating point number only)
- $\backslash\_fp\_ \& \_o:ww \langle floating\ point \rangle \langle floating\ point \rangle$
- $\backslash\_fp\_| \_o:ww \langle floating\ point \rangle \langle floating\ point \rangle$
- $\backslash\_fp\_ternary:NwwN, \backslash\_fp\_ternary\_auxi:NwwN, \backslash\_fp\_ternary\_auxii:NwwN$ have to be understood.

78.2 Tests

$\backslash fp\_if\_exist\_p:N$

$\backslash fp\_if\_exist\_p:c$

$\backslash fp\_if\_exist:N\textcolor{red}{TF}$

$\backslash fp\_if\_exist:c\textcolor{red}{TF}$

Copies of the cs functions defined in l3basics.

26587 $\backslash prg\_new\_eq\_conditional:NNn \backslash fp\_if\_exist:N \backslash cs\_if\_exist:N \{ TF, T, F, p \}$

26588 $\backslash prg\_new\_eq\_conditional:NNn \backslash fp\_if\_exist:c \backslash cs\_if\_exist:c \{ TF, T, F, p \}$

(End of definition for $\backslash fp\_if\_exist:N\textcolor{red}{TF}$. This function is documented on page [270](#).)

`\fp_if_nan_p:n` Evaluate and check if the result is a floating point of the same kind as `nan`.
`\fp_if_nan:nTF`

```

26589 \prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
26590 {
26591   \if:w 3 \exp_last_unbraced:Nf \__fp_kind:w { \__fp_parse:n {#1} }
26592   \prg_return_true:
26593   \else:
26594     \prg_return_false:
26595   \fi:
26596 }

```

(End of definition for `\fp_if_nan:nTF`. This function is documented on page 271.)

78.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we
`\fp_compare:nTF` evaluate #1, then compare with ± 0 . Tuples are true.

```

\__fp_compare_return:w 26597 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
26598 {
26599   \exp_after:wN \__fp_compare_return:w
26600   \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
26601 }
26602 \cs_new:Npn \__fp_compare_return:w #1#2#3\__fp_sep:
26603 {
26604   \if_charcode:w 0
26605     \__fp_if_type_fp:NTwFw
26606     #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
26607     \s__fp 1 \s__fp_stop
26608     \prg_return_false:
26609   \else:
26610     \prg_return_true:
26611   \fi:
26612 }

```

(End of definition for `\fp_compare:nTF` and `\__fp_compare_return:w`. This function is documented on page 271.)

`\fp_compare_p:nNn` Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point
`\fp_compare:nNnTF` numbers swapped to `\__fp_compare_back_any:ww`, defined below. Compare the result
`\__fp_compare_aux:wn` with ‘#2-’, which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.

```

26613 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
26614 {
26615   \if_int_compare:w
26616     \exp_after:wN \__fp_compare_aux:wn
26617     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
26618     = \__fp_int_eval:w ‘#2 - ‘= \__fp_int_eval_end:
26619     \prg_return_true:
26620   \else:
26621     \prg_return_false:
26622   \fi:
26623 }
26624 \cs_new:Npn \__fp_compare_aux:wn #1\__fp_sep: #2
26625 {
26626   \exp_after:wN \__fp_compare_back_any:ww

```

```

26627     \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1\__fp_sep:
26628 }

```

(End of definition for \fp_compare:nNnTF and __fp_compare_aux:wn. This function is documented on page 270.)

```

\__fp_compare_back:ww      \__fp_compare_back_any:ww <y> \__fp_sep: <x> \__fp_sep:
\__fp_bcmp:ww             Expands (in the same way as \int_eval:n) to -1 if  $x < y$ , 0 if  $x = y$ , 1 if  $x > y$ ,
\__fp_compare_back_any:ww and 2 otherwise (denoted as  $x?y$ ). If either operand is nan, stop the comparison with
\__fp_compare_nan:w       \__fp_compare_nan:w returning 2. If  $x$  is negative, swap the outputs 1 and -1 (i.e.,  $>$ 
                           and  $<$ ); we can henceforth assume that  $x \geq 0$ . If  $y \geq 0$ , and they have the same type,
                           either they are normal and we compare them with \__fp_compare_npos:nwnw, or they
                           are equal. If  $y \geq 0$ , but of a different type, the highest type is a larger number. Finally,
                           if  $y \leq 0$ , then  $x > y$ , unless both are zero.
26629 \cs_new:Npn \__fp_compare_back:ww #1#2\__fp_sep: #3#4\__fp_sep:
26630 {
26631   \cs:w
26632     __fp
26633     \__fp_type_from_scan:N #1
26634     _bcmp
26635     \__fp_type_from_scan:N #3
26636     :ww
26637   \cs_end:
26638   #1#2\__fp_sep: #3#4\__fp_sep:
26639 }
26640 \cs_new:Npn \__fp_compare_back_any:ww #1#2\__fp_sep: #3
26641 {
26642   \__fp_if_type_fp:NTwFw
26643   #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
26644   \s__fp \use_ii:nn \s__fp_stop
26645   \__fp_compare_back:ww
26646   {
26647     \cs:w
26648       __fp
26649       \__fp_type_from_scan:N #1
26650       _compare_back
26651       \__fp_type_from_scan:N #3
26652       :ww
26653     \cs_end:
26654   }
26655   #1#2 \__fp_sep: #3
26656 }
26657 \cs_new:Npn \__fp_bcmp:ww
26658   \s__fp \__fp_chk:w #1 #2 #3\__fp_sep:
26659   \s__fp \__fp_chk:w #4 #5 #6\__fp_sep:
26660 {
26661   \int_value:w
26662   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
26663   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
26664   \if_meaning:w 2 #5 - \fi:
26665   \if_meaning:w #2 #5
26666     \if_meaning:w #1 #4
26667       \if_meaning:w 1 #1
26668       \__fp_compare_npos:nwnw #6\__fp_sep: #3\__fp_sep:

```

```

26669         \else:
26670             0
26671         \fi:
26672     \else:
26673         \if_int_compare:w #4 < #1 - \fi: 1
26674     \fi:
26675 \else:
26676     \if_int_compare:w #1#4 = \c_zero_int
26677         0
26678     \else:
26679         1
26680     \fi:
26681 \fi:
26682 \exp_stop_f:
26683 }
26684 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }

```

(End of definition for __fp_compare_back:ww and others.)

```

\__fp_compare_back_tuple:ww Tuple and floating point numbers are not comparable so return 2 in mixed cases or
\__fp_tuple_compare_back:ww when tuples have a different number of items. Otherwise compare pairs of items with
    \__fp_tuple_compare_back_tuple:ww \__fp_compare_back_any:ww and if any don't match return 2 (as \int_value:w 02
    \__fp_tuple_compare_back_loop:w \exp_stop_f:).
26685 \cs_new:Npn \__fp_compare_back_tuple:ww #1\__fp_sep: #2\__fp_sep: { 2 }
26686 \cs_new:Npn \__fp_tuple_compare_back:ww #1\__fp_sep: #2\__fp_sep: { 2 }
26687 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
26688     \s__fp_tuple \__fp_tuple_chk:w #1\__fp_sep:
26689     \s__fp_tuple \__fp_tuple_chk:w #2\__fp_sep:
26690 {
26691     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
26692         { \__fp_array_count:n {#2} }
26693     {
26694         \int_value:w 0
26695         \__fp_tuple_compare_back_loop:w
26696             #1 { \s__fp \prg_break: } \__fp_sep: @
26697             #2 { \s__fp \prg_break: } \__fp_sep:
26698         \prg_break_point:
26699         \exp_stop_f:
26700     }
26701     { 2 }
26702 }
26703 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 \__fp_sep: #3 @ #4#5 \__fp_sep:
26704 {
26705     \use_none:n #1
26706     \use_none:n #4
26707     \if_int_compare:w
26708         \__fp_compare_back_any:ww #1 #2 \__fp_sep: #4 #5 \__fp_sep: = \c_zero_int
26709     \else:
26710         2 \exp_after:wN \prg_break:
26711     \fi:
26712     \__fp_tuple_compare_back_loop:w #3 @
26713 }

```

(End of definition for __fp_compare_back_tuple:ww and others.)

```

\__fp_compare_npos:nwnw \__fp_compare_npos:nwnw {\expo1} \body1 \__fp_sep: {\expo2} \body2
\__fp_compare_significand:nnnnnnnn \__fp_sep:

```

Within an `\int_value:w ... \exp_stop_f:` construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

26714 \cs_new:Npn \__fp_compare_npos:nwnw #1#2\__fp_sep: #3#4\__fp_sep:
26715 {
26716   \if_int_compare:w #1 = #3 \exp_stop_f:
26717     \__fp_compare_significand:nnnnnnnn #2 #4
26718   \else:
26719     \if_int_compare:w #1 < #3 - \fi: 1
26720   \fi:
26721 }
26722 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
26723 {
26724   \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
26725     \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
26726     0
26727   \else:
26728     \if_int_compare:w #3#4 < #7#8 - \fi: 1
26729   \fi:
26730 \else:
26731   \if_int_compare:w #1#2 < #5#6 - \fi: 1
26732 \fi:
26733 }

```

(End of definition for `\__fp_compare_npos:nwnw` and `\__fp_compare_significand:nnnnnnnn`.)

78.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn
26734 \cs_new:Npn \fp_do_until:nn #1#2
26735 {
26736   #2
26737   \fp_compare:nF {#1}
26738   { \fp_do_until:nn {#1} {#2} }
26739 }
26740 \cs_new:Npn \fp_do_while:nn #1#2
26741 {
26742   #2
26743   \fp_compare:nT {#1}
26744   { \fp_do_while:nn {#1} {#2} }
26745 }
26746 \cs_new:Npn \fp_until_do:nn #1#2
26747 {
26748   \fp_compare:nF {#1}
26749   {
26750     #2

```

```

26751         \fp_until_do:nn {#1} {#2}
26752     }
26753 }
26754 \cs_new:Npn \fp_while_do:nn #1#2
26755 {
26756     \fp_compare:nT {#1}
26757     {
26758         #2
26759         \fp_while_do:nn {#1} {#2}
26760     }
26761 }

```

(End of definition for \fp_do_until:nn and others. These functions are documented on page 272.)

\fp_do_until:nNnn As above but not using the nNn syntax.

```

\fp_do_while:nNnn 26762 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
\fp_until_do:nNnn 26763 {
\fp_while_do:nNnn 26764     #4
26765     \fp_compare:nNnF {#1} #2 {#3}
26766     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
26767 }
26768 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
26769 {
26770     #4
26771     \fp_compare:nNnT {#1} #2 {#3}
26772     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
26773 }
26774 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
26775 {
26776     \fp_compare:nNnF {#1} #2 {#3}
26777     {
26778         #4
26779         \fp_until_do:nNnn {#1} #2 {#3} {#4}
26780     }
26781 }
26782 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
26783 {
26784     \fp_compare:nNnT {#1} #2 {#3}
26785     {
26786         #4
26787         \fp_while_do:nNnn {#1} #2 {#3} {#4}
26788     }
26789 }

```

(End of definition for \fp_do_until:nNnn and others. These functions are documented on page 271.)

\fp_step_function:nnnN The approach here is somewhat similar to \int_step_function:nnnN. There are two
\fp_step_function:nnnc subtleties: we use the internal parser __fp_parse:n to avoid converting back and forth
 __fp_step:wwwN from the internal representation; and (due to rounding) even a non-zero step does not
 __fp_step_fp:wwwN guarantee that the loop counter increases.
 __fp_step:NnnnnN

```

26790 \cs_new:Npn \fp_step_function:nnnN #1#2#3
26791 {
26792     \exp_after:wN \__fp_step:wwwN
26793     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}

```



```

26794     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
26795     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
26796   }
26797 \cs_generate_variant:Nn \fp_step_function:nnnN { nnc }

```

Only floating point numbers (not tuples) are allowed arguments. Only “normal” floating points (not ± 0 , $\pm \text{inf}$, nan) can be used as step; if positive, call `\__fp_step:NnnnnN` with argument `>` otherwise `<`. This function has one more argument than its integer counterpart, namely the previous value, to catch the case where the loop has made no progress. Conversion to decimal is done just before calling the user’s function.

```

26798 \cs_new:Npn \__fp_step:wwwN #1#2\__fp_sep: #3#4\__fp_sep: #5#6\__fp_sep: #7
26799 {
26800   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
26801   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
26802   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
26803   \use_i:nnnn
26804   { \__fp_step_fp:wwwN #1#2\__fp_sep: #3#4\__fp_sep: #5#6\__fp_sep: #7 }
26805   \prg_break_point:
26806   \use:n
26807   {
26808     \__fp_error:nfff { step-tuple } { \fp_to_tl:n { #1#2 \__fp_sep: } }
26809     { \fp_to_tl:n { #3#4 \__fp_sep: } } { \fp_to_tl:n { #5#6 \__fp_sep: } }
26810   }
26811 }
26812 \cs_new:Npn \__fp_step_fp:wwwN
26813   #1 \__fp_sep: \s__fp \__fp_chk:w #2#3#4 \__fp_sep: #5\__fp_sep: #6
26814 {
26815   \token_if_eq_meaning:NNTF #2 1
26816   {
26817     \token_if_eq_meaning:NNTF #3 0
26818     { \__fp_step:NnnnnN > }
26819     { \__fp_step:NnnnnN < }
26820   }
26821   {
26822     \token_if_eq_meaning:NNTF #2 0
26823     {
26824       \msg_expandable_error:nnn { kernel }
26825       { zero-step } {#6}
26826     }
26827     {
26828       \__fp_error:nnfn { bad-step } { }
26829       { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 \__fp_sep: } } {#6}
26830     }
26831     \use_none:nnnnn
26832   }
26833   { #1 \__fp_sep: }
26834   { \c_nan_fp }
26835   { \s__fp \__fp_chk:w #2#3#4 \__fp_sep: }
26836   { #5 \__fp_sep: }
26837   #6
26838 }
26839 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
26840 {
26841   \fp_compare:nNnTF {#2} = {#3}

```

```

26842     {
26843         \_fp_error:nffn { tiny-step }
26844         { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
26845     }
26846     {
26847         \fp_compare:nNnF {#2} #1 {#5}
26848         {
26849             \exp_args:Nf #6 { \_fp_to_decimal_dispatch:w #2 }
26850             \_fp_step:NfnnnN
26851             #1 { \_fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
26852         }
26853     }
26854 }
26855 \cs_generate_variant:Nn \_fp_step:NnnnnN { Nf }

```

(End of definition for \fp_step_function:nnnN and others. This function is documented on page 273.)

\fp_step_inline:nnnn As for \int_step_inline:nnnn, create a global function and apply it, following up with
\fp_step_variable:nnnN a break point.

```

\__fp_step:NNnnnn
26856 \cs_new_protected:Npn \fp_step_inline:nnnn
26857 {
26858     \int_gincr:N \g__kernel_prg_map_int
26859     \exp_args:NNc \_fp_step:NNnnnn
26860     \cs_gset_protected:Npn
26861     { \_fp_map_ \int_use:N \g__kernel_prg_map_int :w }
26862 }
26863 \cs_new_protected:Npn \fp_step_variable:nnnN #1#2#3#4#5
26864 {
26865     \int_gincr:N \g__kernel_prg_map_int
26866     \exp_args:NNc \_fp_step:NNnnnn
26867     \cs_gset_protected:Npe
26868     { \_fp_map_ \int_use:N \g__kernel_prg_map_int :w }
26869     {#1} {#2} {#3}
26870     {
26871         \tl_set:Nn \exp_not:N #4 {##1}
26872         \exp_not:n {#5}
26873     }
26874 }
26875 \cs_new_protected:Npn \_fp_step:NNnnnn #1#2#3#4#5#6
26876 {
26877     #1 #2 ##1 {#6}
26878     \fp_step_function:nnnN {#3} {#4} {#5} #2
26879     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
26880 }

```

(End of definition for \fp_step_inline:nnnn, \fp_step_variable:nnnN, and _fp_step:NNnnnn. These functions are documented on page 273.)

```

26881 \msg_new:nnn { fp } { step-tuple }
26882 { Tuple~argument~in~fp_step...~{#1}{#2}{#3}. }
26883 \msg_new:nnn { fp } { bad-step }
26884 { Invalid~step~size~#2~for~function~#3. }
26885 \msg_new:nnn { fp } { tiny-step }
26886 { Tiny~step~size~( #1 + #2 = #1 ) ~for~function~#3. }

```

78.5 Extrema

`__fp_minmax_o:Nw`
`__fp_minmax_aux_o:Nw`

First check all operands are floating point numbers. The argument #1 is 2 to find the maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

26887 \cs_new:Npn __fp_minmax_o:Nw #1
26888 {
26889   __fp_parse_function_all_fp_o:fnw
26890   { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
26891   { __fp_minmax_aux_o:Nw #1 }
26892 }
26893 \cs_new:Npn __fp_minmax_aux_o:Nw #1#2 @
26894 {
26895   \if_meaning:w 0 #1
26896     \exp_after:wN __fp_minmax_loop:Nww \exp_after:wN +
26897   \else:
26898     \exp_after:wN __fp_minmax_loop:Nww \exp_after:wN -
26899   \fi:
26900   #2
26901   \s_fp __fp_chk:w 2 #1 \s_fp_exact __fp_sep:
26902   \s_fp __fp_chk:w { 3 __fp_minmax_break_o:w } __fp_sep:
26903 }

```

(End of definition for `__fp_minmax_o:Nw` and `__fp_minmax_aux_o:Nw`.)

`__fp_minmax_loop:Nww`

The first argument is `-` or `+` to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```

26904 \cs_new:Npn __fp_minmax_loop:Nww
26905   #1 \s_fp __fp_chk:w #2#3__fp_sep: \s_fp __fp_chk:w #4#5__fp_sep:
26906   {
26907     \if_meaning:w 3 #4
26908       \if_meaning:w 3 #2
26909         __fp_minmax_auxi:ww
26910       \else:
26911         __fp_minmax_auxii:ww
26912       \fi:
26913     \else:
26914       \if_int_compare:w
26915         __fp_compare_back:ww
26916         \s_fp __fp_chk:w #4#5__fp_sep:
26917         \s_fp __fp_chk:w #2#3__fp_sep:
26918         = #1 1 \exp_stop_f:
26919         __fp_minmax_auxii:ww

```

```

26920         \else:
26921             \__fp_minmax_auxi:ww
26922         \fi:
26923     \fi:
26924     \__fp_minmax_loop:Nww #1
26925         \s__fp \__fp_chk:w #2#3\__fp_sep:
26926         \s__fp \__fp_chk:w #4#5\__fp_sep:
26927 }

```

(End of definition for __fp_minmax_loop:Nww.)

__fp_minmax_auxi:ww Keep the first/second number, and remove the other.

```

\__fp_minmax_auxii:ww
26928 \cs_new:Npn \__fp_minmax_auxi:ww
26929     #1 \fi: \fi: #2 \s__fp #3 \__fp_sep: \s__fp #4\__fp_sep:
26930     { \fi: \fi: #2 \s__fp #3 \__fp_sep: }
26931 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 \__fp_sep:
26932     { \fi: \fi: #2 }

```

(End of definition for __fp_minmax_auxi:ww and __fp_minmax_auxii:ww.)

__fp_minmax_break_o:w This function is called from within an \if_meaning:w test. Skip to the end of the tests, close the current test with \fi:, clean up, and return the appropriate number with one post-expansion.

```

26933 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3\__fp_sep: #4\__fp_sep:
26934     { \fi: \__fp_exp_after_o:w \s__fp #3\__fp_sep: }

```

(End of definition for __fp_minmax_break_o:w.)

78.6 Boolean operations

__fp_not_o:w Return true or false, with two expansions, one to exit the conditional, and one to please l3fp-parse. The first argument is provided by l3fp-parse and is ignored.

```

\__fp_tuple_not_o:w
26935 \cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
26936     {
26937         \if_meaning:w 0 #2
26938             \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
26939         \else:
26940             \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
26941         \fi:
26942     }
26943 \cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }

```

(End of definition for __fp_not_o:w and __fp_tuple_not_o:w.)

__fp_&_o:w For and, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For or, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking __fp_&_o:w, inserting an extra argument, \else:, before \s__fp. In all cases, expand after the floating point number.

```

\__fp_tuple_&_o:w
\__fp_&_tuple_o:w
\__fp_tuple_&_tuple_o:w
\__fp_|_o:w
\__fp_tuple_|_o:w
\__fp_|_tuple_o:w
\__fp_tuple_|_tuple_o:w
\__fp_and_return:wNw
26944 \group_begin:
26945     \char_set_catcode_letter:N &
26946     \char_set_catcode_letter:N |
26947     \cs_new:Npn \__fp_&_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep:

```

```

26948 {
26949     \if_meaning:w 0 #2 #1
26950     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3\__fp_sep:
26951     \fi:
26952     \__fp_exp_after_o:w
26953 }
26954 \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3\__fp_sep:
26955 {
26956     \if_meaning:w 0 #2 #1
26957     \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3\__fp_sep:
26958     \fi:
26959     \__fp_exp_after_tuple_o:w
26960 }
26961 \cs_new:Npn \__fp_tuple_&_o:ww #1\__fp_sep: { \__fp_exp_after_o:w }
26962 \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1\__fp_sep: { \__fp_exp_after_tuple_o:w }
26963 \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
26964 \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
26965 \cs_new:Npn \__fp_tuple_|_o:ww #1\__fp_sep: #2\__fp_sep:
26966 { \__fp_exp_after_tuple_o:w #1\__fp_sep: }
26967 \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1\__fp_sep: #2\__fp_sep:
26968 { \__fp_exp_after_tuple_o:w #1\__fp_sep: }
26969 \group_end:
26970 \cs_new:Npn \__fp_and_return:wNw #1\__fp_sep: \fi: #2\__fp_sep:
26971 { \fi: \__fp_exp_after_o:w #1\__fp_sep: }

```

(End of definition for __fp_&_o:ww and others.)

78.7 Ternary operator

```

\__fp_ternary:NwN
\__fp_ternary_auxi:NwN
\__fp_ternary_auxii:NwN

```

The first function receives the test and the true branch of the ?: ternary operator. It calls __fp_ternary_auxii:NwN if the test branch is a floating point number ± 0 , and otherwise calls __fp_ternary_auxi:NwN. These functions select one of their two arguments.

```

26972 \cs_new:Npn \__fp_ternary:NwN #1 #2#3@ #4@ #5
26973 {
26974     \if_meaning:w \__fp_parse_infix_:N #5
26975     \if_charcode:w 0
26976         \__fp_if_type_fp:NTwFw
26977         #2 { \use_i:nn \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
26978         \s__fp 1 \s__fp_stop
26979         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwN
26980     \else:
26981         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwN
26982     \fi:
26983     \exp_after:wN #1
26984     \exp:w \exp_end_continue_f:w
26985     \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
26986     \exp_after:wN @
26987     \exp:w
26988         \__fp_parse_operand:Nw \c__fp_prec_colon_int
26989         \__fp_parse_expand:w
26990 \else:
26991     \msg_expandable_error:nnnn

```

```

26992     { fp } { missing } { : } { ~for~?: }
26993     \exp_after:wN \__fp_parse_continue:NwN
26994     \exp_after:wN #1
26995     \exp:w \exp_end_continue_f:w
26996     \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
26997     \exp_after:wN #5
26998     \exp_after:wN #1
26999     \fi:
27000   }
27001   \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
27002   {
27003     \exp_after:wN \__fp_parse_continue:NwN
27004     \exp_after:wN #1
27005     \exp:w \exp_end_continue_f:w
27006     \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
27007     #4 #1
27008   }
27009   \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
27010   {
27011     \exp_after:wN \__fp_parse_continue:NwN
27012     \exp_after:wN #1
27013     \exp:w \exp_end_continue_f:w
27014     \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
27015     #4 #1
27016   }

```

(End of definition for __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, and __fp_ternary_auxii:NwwN.)

```

27017 </code>

```

Chapter 79

l3fp-basics implementation

```
27018 ⟨*code⟩
27019 ⟨@@=fp⟩
```

The `l3fp-basics` module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```

__fp_parse_word_abs:N
__fp_parse_word_logb:N
__fp_parse_word_sign:N
__fp_parse_word_sqrt:N
27020 \cs_new:Npn \__fp_parse_word_abs:N
27021   { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
27022 \cs_new:Npn \__fp_parse_word_logb:N
27023   { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
27024 \cs_new:Npn \__fp_parse_word_sign:N
27025   { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
27026 \cs_new:Npn \__fp_parse_word_sqrt:N
27027   { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(End of definition for __fp_parse_word_abs:N and others.)

79.1 Addition and subtraction

We define here two functions, `\__fp_-_o:ww` and `\__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `\__fp_add_big_i_o:wNww`, is used in `l3fp-expo`.

The logic goes as follows:

- `\__fp_-_o:ww` calls `\__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `\__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;

- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `\__fp-basics_pack...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

79.1.1 Sign, exponent, and special numbers

`\__fp_-_o:ww` The `\__fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `\__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `\__fp+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```

27028 \cs_new:cpe { __fp_-_o:ww } \s__fp
27029 {
27030   \exp_not:c { __fp+_o:ww }
27031   \exp_not:n { \s__fp \__fp_neg_sign:N }
27032 }

```

(End of definition for `\__fp_-_o:ww`.)

`\__fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `\__fp_-_o:ww` with `\__fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the `<types>` #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked `<sign2>` (expansion of #1#5) as an argument. If the `<types>` are distinct, the result is simply the floating point number with the highest `<type>`. Since case 3 (used for two nan) also picks the first operand, we can also use it when `<type1>` is greater than `<type2>`. Also note that we don't need to worry about `<sign2>` in that case since the second operand is discarded.

```

27033 \cs_new:cpn { __fp+_o:ww }
27034   \s__fp #1 \__fp_chk:w #2 #3 \__fp_sep: \s__fp \__fp_chk:w #4 #5
27035 {
27036   \if_case:w
27037     \if_meaning:w #2 #4
27038       #2
27039     \else:
27040       \if_int_compare:w #2 > #4 \exp_stop_f:
27041         3
27042       \else:
27043         4
27044       \fi:
27045     \fi:
27046   \exp_stop_f:
27047     \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
27048   \or:   \exp_after:wN \__fp_add_normal_o:Nww \int_value:w

```



```

27049 \or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
27050 \or: \__fp_case_return_i_o:ww
27051 \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
27052 \fi:
27053 #1 #5
27054 \s__fp \__fp_chk:w #2 #3 \__fp_sep:
27055 \s__fp \__fp_chk:w #4 #5
27056 }

```

(End of definition for __fp_+_o:ww.)

__fp_add_return_ii_o:Nww Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

27057 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 \__fp_sep: \s__fp \__fp_chk:w #3 #4
27058 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }

```

(End of definition for __fp_add_return_ii_o:Nww.)

__fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

27059 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
27060 {
27061   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
27062   \exp_after:wN \__fp_add_return_ii_o:Nww
27063   \else:
27064     \__fp_case_return_i_o:ww
27065   \fi:
27066   #1
27067   \s__fp \__fp_chk:w 0 #2
27068 }

```

(End of definition for __fp_add_zeros_o:Nww.)

__fp_add_inf_o:Nww If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2 \rangle$ (#1) and the $\langle sign_2 \rangle$ (#4) are identical.

```

27069 \cs_new:Npn \__fp_add_inf_o:Nww
27070   #1 \s__fp \__fp_chk:w 2 #2 #3 \__fp_sep: \s__fp \__fp_chk:w 2 #4
27071 {
27072   \if_meaning:w #1 #2
27073   \__fp_case_return_i_o:ww
27074   \else:
27075     \__fp_case_use:nw
27076     {
27077       \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
27078       { \token_if_eq_meaning:NNTF #1 #4 + - }
27079     }
27080   \fi:
27081   \s__fp \__fp_chk:w 2 #2 #3 \__fp_sep:
27082   \s__fp \__fp_chk:w 2 #4
27083 }

```

(End of definition for __fp_add_inf_o:Nww.)

```

\__fp_add_normal_o:Nww      \__fp_add_normal_o:Nww <sign2> \s__fp \__fp_chk:w 1 <sign1> <exp1>
                             <body1> \__fp_sep: \s__fp \__fp_chk:w 1 <initial sign2> <exp2> <body2>
                             \__fp_sep:

```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

27084 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
27085 {
27086   \if_meaning:w #1#2
27087     \exp_after:wN \__fp_add_npos_o:NnwNnw
27088   \else:
27089     \exp_after:wN \__fp_sub_npos_o:NnwNnw
27090   \fi:
27091   #2
27092 }

```

(End of definition for __fp_add_normal_o:Nww.)

79.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```

\__fp_add_npos_o:NnwNnw      \__fp_add_npos_o:NnwNnw <sign1> <exp1> <body1> \__fp_sep: \s__fp \__fp_-
                             chk:w 1 <initial sign2> <exp2> <body2> \__fp_sep:

```

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an $\backslash\_fp\_int\_eval:w$, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to $\backslash\_fp\_sanitize:Nw$ which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by $\backslash\_fp\_add\_big\_i:wNww$ or $\backslash\_fp\_add\_big\_ii:wNww$. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

27093 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 \__fp_sep: \s__fp \__fp_chk:w 1 #4 #5
27094 {
27095   \exp_after:wN \__fp_sanitize:Nw
27096   \exp_after:wN #1
27097   \int_value:w \__fp_int_eval:w
27098   \if_int_compare:w #2 > #5 \exp_stop_f:
27099     #2
27100   \exp_after:wN \__fp_add_big_i_o:wNww \int_value:w -
27101   \else:
27102     #5
27103   \exp_after:wN \__fp_add_big_ii_o:wNww \int_value:w
27104   \fi:
27105   \__fp_int_eval:w #5 - #2 \__fp_sep: #1 #3 \__fp_sep:
27106 }

```

(End of definition for __fp_add_npos_o:NnwNnw.)

```

\__fp_add_big_i_o:wNww      \__fp_add_big_i_o:wNww <shift> \__fp_sep: <final sign> <body1> \__fp_-
\__fp_add_big_ii_o:wNww     sep: <body2> \__fp_sep:

```

Used in `l3fp-expo`. Shift the significand of the small number, then add with $\backslash\_fp\_add\_significand_o:NnnwnnnnN$.

```

27107 \cs_new:Npn \__fp_add_big_i_o:wNww #1 \__fp_sep: #2 #3 \__fp_sep: #4 \__fp_sep:
27108 {

```

```

27109     \__fp_decimate:nNnnnn {#1}
27110     \__fp_add_significand_o:NnnwnnnnN
27111     #4
27112     #3
27113     #2
27114   }
27115 \cs_new:Npn \__fp_add_big_ii_o:wNww #1\__fp_sep: #2 #3\__fp_sep: #4\__fp_sep:
27116 {
27117     \__fp_decimate:nNnnnn {#1}
27118     \__fp_add_significand_o:NnnwnnnnN
27119     #3
27120     #4
27121     #2
27122   }

```

(End of definition for __fp_add_big_i_o:wNww and __fp_add_big_ii_o:wNww.)

```

\__fp_add_significand_o:NnnwnnnnN     \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\__fp_add_significand_pack:NNNNNNN    <extra-digits> \__fp_sep: {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\dots95 \rightarrow 1.00\dots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

27123 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4\__fp_sep: #5#6#7#8
27124 {
27125     \exp_after:wN \__fp_add_significand_test_o:N
27126     \int_value:w \__fp_int_eval:w 1#5#6 + #2
27127     \exp_after:wN \__fp_add_significand_pack:NNNNNNN
27128     \int_value:w \__fp_int_eval:w 1#7#8 + #3 \__fp_sep: #1
27129   }
27130 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
27131 {
27132     \if_meaning:w 2 #1
27133     + 1
27134     \fi:
27135     \__fp_sep: #2 #3 #4 #5 #6 #7 \__fp_sep:
27136   }
27137 \cs_new:Npn \__fp_add_significand_test_o:N #1
27138 {
27139     \if_meaning:w 2 #1
27140     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
27141     \else:
27142     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
27143     \fi:
27144   }

```

(End of definition for __fp_add_significand_o:NnnwnnnnN, __fp_add_significand_pack:NNNNNNN, and __fp_add_significand_test_o:N.)

```

\__fp_add_significand_no_carry_o:wwwNN     \__fp_add_significand_no_carry_o:wwwNN <8d> \__fp_sep: <6d> \__fp_-
                                          sep: <2d> \__fp_sep: <rounding digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function __fp_basics_pack_high:NNNNNw takes care of the case where rounding brings a carry.

```

27145 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
27146   #1\__fp_sep: #2\__fp_sep: #3#4 \__fp_sep: #5#6
27147   {
27148     \exp_after:wN \__fp_basics_pack_high:NNNNNw
27149     \int_value:w \__fp_int_eval:w 1 #1
27150     \exp_after:wN \__fp_basics_pack_low:NNNNNw
27151     \int_value:w \__fp_int_eval:w 1 #2 #3#4
27152     + \__fp_round:NNN #6 #4 #5
27153     \exp_after:wN \__fp_sep:
27154   }

```

(End of definition for __fp_add_significand_no_carry_o:wwwNN.)

```

\__fp_add_significand_carry_o:wwwNN <8d> \__fp_sep: <6d> \__fp_-
sep: <2d> \__fp_sep: <rounding digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

27155 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
27156   #1\__fp_sep: #2\__fp_sep: #3#4\__fp_sep: #5#6
27157   {
27158     + 1
27159     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
27160     \int_value:w \__fp_int_eval:w 1 1 #1
27161     \exp_after:wN \__fp_basics_pack_weird_low:NNNNNw
27162     \int_value:w \__fp_int_eval:w 1 #2#3 +
27163     \exp_after:wN \__fp_round:NNN
27164     \exp_after:wN #6
27165     \exp_after:wN #3
27166     \int_value:w \__fp_round_digit:Nw #4 #5 \__fp_sep:
27167     \exp_after:wN \__fp_sep:
27168   }

```

(End of definition for __fp_add_significand_carry_o:wwwNN.)

79.1.3 Absolute subtraction

```

\__fp_sub_npos_o:NnwNnw \__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> \__fp_sep: \s__fp \__fp_-
\__fp_sub_eq_o:Nwnnw \__fp_sub_eq_o:Nwnnw chk:w 1 <initial sign2> <exp2> <body2> \__fp_sep:
\__fp_sub_npos_ii_o:Nwnnw \__fp_sub_npos_ii_o:Nwnnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call __fp_sub_npos_i_o:Nwnnw with the opposite of <sign₁>.

```

27169 \cs_new:Npn \__fp_sub_npos_o:NnwNnw
27170   #1#2#3\__fp_sep: \s__fp \__fp_chk:w 1 #4#5#6\__fp_sep:
27171   {
27172     \if_case:w
27173       \__fp_compare_npos:nwnw {#2} #3\__fp_sep: {#5} #6\__fp_sep: \exp_stop_f:
27174       \exp_after:wN \__fp_sub_eq_o:Nwnnw
27175     \or:
27176       \exp_after:wN \__fp_sub_npos_i_o:Nwnnw
27177     \else:
27178       \exp_after:wN \__fp_sub_npos_ii_o:Nwnnw
27179     \fi:

```

```

27180     #1 {#2} #3\_fp_sep: {#5} #6\_fp_sep:
27181   }
27182   \cs_new:Npn \_fp_sub_eq_o:Nnwnw #1#2\_fp_sep: #3\_fp_sep:
27183     { \exp_after:wN \c_zero_fp }
27184   \cs_new:Npn \_fp_sub_npos_ii_o:Nnwnw #1 #2\_fp_sep: #3\_fp_sep:
27185     {
27186       \exp_after:wN \_fp_sub_npos_i_o:Nnwnw
27187       \int_value:w \_fp_neg_sign:N #1
27188       #3\_fp_sep: #2\_fp_sep:
27189     }

```

(End of definition for _fp_sub_npos_o:Nnwnw, _fp_sub_eq_o:Nnwnw, and _fp_sub_npos_ii_o:Nnwnw.)

_fp_sub_npos_i_o:Nnwnw After the computation is done, _fp_sanitize:Nw checks for overflow/underflow. It expects the *final sign* and the *exponent* (delimited by _fp_sep:). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the *near* auxiliary. Otherwise, decimate *y*, then call the *far* auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

27190   \cs_new:Npn \_fp_sub_npos_i_o:Nnwnw #1 #2#3\_fp_sep: #4#5\_fp_sep:
27191     {
27192       \exp_after:wN \_fp_sanitize:Nw
27193       \exp_after:wN #1
27194       \int_value:w \_fp_int_eval:w
27195       #2
27196       \if_int_compare:w #2 = #4 \exp_stop_f:
27197         \exp_after:wN \_fp_sub_back_near_o:nnnnnnnnN
27198       \else:
27199         \exp_after:wN \_fp_decimate:nNnnnn \exp_after:wN
27200         { \int_value:w \_fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
27201         \exp_after:wN \_fp_sub_back_far_o:NnnwnnnnnN
27202       \fi:
27203       #5
27204       #3
27205       #1
27206     }

```

(End of definition for _fp_sub_npos_i_o:Nnwnw.)

_fp_sub_back_near_o:nnnnnnnnN _fp_sub_back_near_o:nnnnnnnnN {<Y₁>} {<Y₂>} {<Y₃>} {<Y₄>} {<X₁>}
_fp_sub_back_near_pack:Nnnnnnw {<X₂>} {<X₃>} {<X₄>} {<final sign>}
_fp_sub_back_near_after:wNNNNw

In this case, the subtraction is exact, so we discard the *final sign* #9. The very large shifts of 10⁹ and 1.1·10⁹ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

27207   \cs_new:Npn \_fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
27208     {
27209       \exp_after:wN \_fp_sub_back_near_after:wNNNNw
27210       \int_value:w \_fp_int_eval:w 10#5#6 - #1#2 - 11
27211       \exp_after:wN \_fp_sub_back_near_pack:Nnnnnnw
27212       \int_value:w \_fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN \_fp_sep:
27213     }

```

```

27214 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 \__fp_sep:
27215 { + #1#2 \__fp_sep: {#3#4#5#6} {#7} \__fp_sep: }
27216 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 \__fp_sep:
27217 {
27218   \if_meaning:w 0 #1
27219     \exp_after:wN \__fp_sub_back_shift:wnnnn
27220   \fi:
27221   \__fp_sep: {#1#2#3#4} {#5}
27222 }

```

(End of definition for __fp_sub_back_near_o:nnnnnnnnN, __fp_sub_back_near_pack:NNNNNNw, and __fp_sub_back_near_after:wNNNNw.)

```

\__fp_sub_back_shift:wnnnn      \__fp_sub_back_shift:wnnnn \__fp_sep: {\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle}
\__fp_sub_back_shift_ii:ww      \__fp_sep:
  \__fp_sub_back_shift_iii:NNNNNNNNw
    \__fp_sub_back_shift_iv:nnnnw

```

This function is called with $\langle Z_1 \rangle \leq 999$. Act with `\number` to trim leading zeros from $\langle Z_1 \rangle$ $\langle Z_2 \rangle$ (we don't do all four blocks at once, since non-zero blocks would then overflow TeX's integers). If the first two blocks are zero, the auxiliary receives an empty `#1` and trims `#2#30` from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from `#1` alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from `#1#2#3` (when `#1` is empty, the space before `#2#3` is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

27223 \cs_new:Npn \__fp_sub_back_shift:wnnnn \__fp_sep: #1#2
27224 {
27225   \exp_after:wN \__fp_sub_back_shift_ii:ww
27226   \int_value:w #1 #2 0 \__fp_sep:
27227 }
27228 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 \__fp_sep: #2#3 \__fp_sep:
27229 {
27230   \if_meaning:w @ #1 @
27231     - 7
27232   - \exp_after:wN \use_i:nnn
27233     \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
27234     \int_value:w #2#3 0 ~ 123456789\__fp_sep:
27235   \else:
27236     - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789\__fp_sep:
27237   \fi:
27238   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNNN
27239   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNNN
27240   \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
27241   \exp_after:wN \__fp_sep:
27242   \int_value:w
27243   #1 ~ #2#3 0 ~ 0000 0000 0000 000 \__fp_sep:
27244 }
27245 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9\__fp_sep: {#8}
27246 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 \__fp_sep: #2 \__fp_sep:
27247 { \__fp_sep: #1 \__fp_sep: }

```

(End of definition for __fp_sub_back_shift:wnnnn and others.)

```

\__fp_sub_back_far_o:NnnwnnnnnN \__fp_sub_back_far_o:NnnwnnnnnN \langle rounding \rangle {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\langle extra-digits \rangle \__fp_sep: {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} \langle final sign \rangle

```

If the difference is greater than $10^{\langle \text{expo}_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle \text{expo}_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and `\__fp_sep:` delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `\__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `\__fp_sep:` delimiter).

```

27248 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4\__fp_sep: #5#6#7#8
27249 {
27250   \if_case:w
27251     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
27252     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
27253     0
27254   \else:
27255     \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
27256   \fi:
27257   \else:
27258     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
27259   \fi:
27260   \exp_stop_f:
27261     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
27262   \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNNN
27263   \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNNN
27264   \fi:
27265   #2 ~ #3 \__fp_sep: #5 #6 ~ #7 #8 \__fp_sep: #1
27266 }

```

(End of definition for `\__fp_sub_back_far_o:NnnwnnnnN`.)

`\__fp_sub_back_quite_far_o:wwNN`
`\__fp_sub_back_quite_far_ii:NN`

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the `\langle rounding \rangle #3` and the `\langle final sign \rangle #4` control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the `\langle rounding \rangle` digit is less than or equal to 5 (remember that the `\langle rounding \rangle` digit is only equal to 5 if there was no further non-zero digit).

```

27267 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1\__fp_sep: #2\__fp_sep: #3#4
27268 {
27269   \exp_after:wN \__fp_sub_back_quite_far_ii:NN
27270   \exp_after:wN #3
27271   \exp_after:wN #4
27272 }
27273 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
27274 {
27275   \if_case:w \__fp_round_neg:NNN #2 0 #1
27276     \exp_after:wN \use_i:nn
27277   \else:
27278     \exp_after:wN \use_ii:nn
27279   \fi:
27280   { \__fp_sep: {1000} {0000} {0000} {0000} \__fp_sep: }
27281   { - 1 \__fp_sep: {9999} {9999} {9999} {9999} \__fp_sep: }
27282 }

```

(End of definition for `\__fp_sub_back_quite_far_o:wwNN` and `\__fp_sub_back_quite_far_ii:NN`.)

`\__fp_sub_back_not_far_o:wwwNNN`

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with `-1`). Then proceed in a way

similar to the `near` auxiliaries seen earlier, but multiplying x by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if `\__fp_round_neg:NNN` returns 1. This function expects the *final sign* #6, the last digit of `1100000000+#40-#2`, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that `\__fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of #2.

```

27283 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2\__fp_sep: #3 ~ #4\__fp_sep: #5#6
27284 {
27285   - 1
27286   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
27287   \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
27288   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
27289   \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
27290   - \exp_after:wN \__fp_round_neg:NNN
27291   \exp_after:wN #6
27292   \use_none:nnnnnnn #2 #5
27293   \exp_after:wN \__fp_sep:
27294 }

```

(End of definition for `\__fp_sub_back_not_far_o:wwwNN`.)

`\__fp_sub_back_very_far_o:wwwNN`
`\__fp_sub_back_very_far_ii_o:nnNwwNN`

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *rounding* digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

27295 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
27296 {
27297   \__fp_pack_eight:wNNNNNNNN
27298   \__fp_sub_back_very_far_ii_o:nnNwwNN
27299   { 0 #1#2#3 #4#5#6#7 }
27300   \__fp_sep:
27301 }
27302 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN
27303   #1#2 \__fp_sep: #3 \__fp_sep: #4 ~ #5\__fp_sep: #6#7
27304 {
27305   \exp_after:wN \__fp_basics_pack_high:NNNNNw
27306   \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
27307   \exp_after:wN \__fp_basics_pack_low:NNNNNw
27308   \int_value:w \__fp_int_eval:w 2#5 - #2
27309   - \exp_after:wN \__fp_round_neg:NNN
27310   \exp_after:wN #7
27311   \int_value:w
27312   \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
27313     1 \else: 2 \fi:
27314   \int_value:w \__fp_round_digit:Nw #3 #6 \__fp_sep:
27315   \exp_after:wN \__fp_sep:
27316 }

```

(End of definition for `\__fp_sub_back_very_far_o:wwwNN` and `\__fp_sub_back_very_far_ii_o:nnNwwNN`.)

79.2 Multiplication

79.2.1 Signs, and special numbers

`\__fp*_o:ww` We go through an auxiliary, which is common with `\__fp/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `\__fp/_o:ww`.

```

27317 \cs_new:cpn { __fp*_o:ww }
27318 {
27319     \__fp_mul_cases_o:NnNnw
27320     *
27321     { - 2 + }
27322     \__fp_mul_npos_o:Nww
27323     { }
27324 }
```

(End of definition for `\__fp*_o:ww`.)

`\__fp_mul_cases_o:NnNnw` Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `\__fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0 \times \infty$ and $\infty \times 0$ to case 6 or 7 (invalid operation), and the products $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```

27325 \cs_new:Npn \__fp_mul_cases_o:NnNnw
27326     #1#2#3#4 \s__fp \__fp_chk:w #5#6#7\__fp_sep: \s__fp \__fp_chk:w #8#9
27327 {
27328     \if_case:w \__fp_int_eval:w
27329         \if_int_compare:w #5 #8 = 11 ~
27330             1
27331         \else:
27332             \if_meaning:w 3 #8
27333             3
27334         \else:
27335             \if_meaning:w 3 #5
27336             2
27337         \else:
27338             \if_int_compare:w #5 #8 = 10 ~
27339             9 #2 - 2
27340         \else:
27341             (#5 #2 #8) / 2 * 2 + 7
27342         \fi:
27343     \fi:
27344     \fi:
27345     \if_meaning:w #6 #9 - 1 \fi:
27346 }
```

```

27347         \__fp_int_eval_end:
27348         \__fp_case_use:nw { #3 0 }
27349     \or: \__fp_case_use:nw { #3 2 }
27350     \or: \__fp_case_return_i_o:ww
27351     \or: \__fp_case_return_ii_o:ww
27352     \or: \__fp_case_return_o:Nww \c_zero_fp
27353     \or: \__fp_case_return_o:Nww \c_minus_zero_fp
27354     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
27355     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
27356     \or: \__fp_case_return_o:Nww \c_inf_fp
27357     \or: \__fp_case_return_o:Nww \c_minus_inf_fp
27358     #4
27359     \fi:
27360     \s__fp \__fp_chk:w #5 #6 #7 \__fp_sep:
27361     \s__fp \__fp_chk:w #8 #9
27362 }

```

(End of definition for __fp_mul_cases_o:nNnnww.)

79.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign1> {<exp1>}
<body1> \__fp_sep: \s__fp \__fp_chk:w 1 <sign2> {<exp2>} <body2> \__fp_-
sep:

```

After the computation, __fp_sanitize:Nw checks for overflow or underflow. As we did for addition, __fp_int_eval:w computes the exponent, catching any shift coming from the computation in the significand. The <final sign> is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by __fp_mul_significand_o:nnnnNnnnn.

This is also used in l3fp-convert.

```

27363 \cs_new:Npn \__fp_mul_npos_o:Nww
27364   #1 \s__fp \__fp_chk:w #2 #3 #4 #5 \__fp_sep:
27365   \s__fp \__fp_chk:w #6 #7 #8 #9 \__fp_sep:
27366 {
27367   \exp_after:wN \__fp_sanitize:Nw
27368   \exp_after:wN #1
27369   \int_value:w \__fp_int_eval:w
27370   #4 + #8
27371   \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
27372 }

```

(End of definition for __fp_mul_npos_o:Nww.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
\__fp_mul_significand_drop:NNNNNw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_keep:NNNNNw

```

Note the three __fp_sep:s at the end of the definition. One is for the last __fp_mul_significand_drop:NNNNNw; one is for __fp_round_digit:Nw later on; and one, preceded by \exp_after:wN, which is correctly expanded (within an __fp_int_eval:w), is used by __fp_basics_pack_low:NNNNNw.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number

of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `\__fp_int_eval:w`.

```

27373 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
27374 {
27375   \exp_after:wN \__fp_mul_significand_test_f:NNN
27376   \exp_after:wN #5
27377   \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +
27378   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
27379   \int_value:w \__fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
27380   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
27381   \int_value:w \__fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
27382   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
27383   \int_value:w \__fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
27384   #3*#7 + #4*#6 +
27385   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
27386   \int_value:w \__fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
27387   #4*#7 +
27388   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
27389   \int_value:w \__fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
27390   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
27391   \int_value:w \__fp_int_eval:w 100000000 + #4*#9 \__fp_sep:
27392   \__fp_sep: \exp_after:wN \__fp_sep:
27393 }
27394 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6\__fp_sep:
27395 { #1#2#3#4#5 \__fp_sep: + #6 }
27396 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6\__fp_sep:
27397 { #1#2#3#4#5 \__fp_sep: #6 \__fp_sep: }

```

(End of definition for `\__fp_mul_significand_o:nnnnNnnnn`, `\__fp_mul_significand_drop:NNNNNw`, and `\__fp_mul_significand_keep:NNNNNw`.)

```

\__fp_mul_significand_test_f:NNN   \__fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> \__fp_sep:
<digits 9-12> \__fp_sep: <digits 13-16> \__fp_sep: + <digits 17-20>
+ <digits 21-24> + <digits 25-28> + <digits 29-32> \__fp_sep: \exp_-
after:wN \__fp_sep:

```

If the *<digit 1>* is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if *<digit 1>* is zero, we care about digits 17 and 18, and whether further digits are zero.

```

27398 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
27399 {
27400   \if_meaning:w 0 #3
27401   \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
27402   \else:
27403   \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
27404   \fi:
27405   #1 #3
27406 }

```

(End of definition for `\__fp_mul_significand_test_f:NNN`.)

`\__fp_mul_significand_large_f:NwwNNNN` In this branch, *<digit 1>* is non-zero. The result is thus *<digits 1-16>*, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are

zero or not. Here, `\__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a *<rounding digit>*, suitable for `\__fp_round:NNN`.

```

27407 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN
27408   #1 #2\__fp_sep: #3\__fp_sep: #4#5#6#7\__fp_sep: +
27409   {
27410     \exp_after:wN \__fp_basics_pack_high:NNNNNw
27411     \int_value:w \__fp_int_eval:w 1#2
27412     \exp_after:wN \__fp_basics_pack_low:NNNNNw
27413     \int_value:w \__fp_int_eval:w 1#3#4#5#6#7
27414     + \exp_after:wN \__fp_round:NNN
27415     \exp_after:wN #1
27416     \exp_after:wN #7
27417     \int_value:w \__fp_round_digit:Nw
27418   }

```

(End of definition for `\__fp_mul_significand_large_f:NwwNNNN`.)

`\__fp_mul_significand_small_f:NNwwN`

In this branch, *<digit 1>* is zero. Our result is thus *<digits 2-17>*, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

27419 \cs_new:Npn \__fp_mul_significand_small_f:NNwwN
27420   #1 #2#3\__fp_sep: #4#5\__fp_sep: #6\__fp_sep: + #7
27421   {
27422     - 1
27423     \exp_after:wN \__fp_basics_pack_high:NNNNNw
27424     \int_value:w \__fp_int_eval:w 1#3#4
27425     \exp_after:wN \__fp_basics_pack_low:NNNNNw
27426     \int_value:w \__fp_int_eval:w 1#5#6#7
27427     + \exp_after:wN \__fp_round:NNN
27428     \exp_after:wN #1
27429     \exp_after:wN #7
27430     \int_value:w \__fp_round_digit:Nw
27431   }

```

(End of definition for `\__fp_mul_significand_small_f:NNwwN`.)

79.3 Division

79.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`\__fp/_o:ww`

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `\__fp_div_npos_o:Nww` rather than `\__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `\__fp_mul_cases_o:NnNww` are provided as the fourth argument here.

```

27432 \cs_new:cpn { __fp/_o:ww }
27433   {

```

```

27434 \__fp_mul_cases_o:NnNnw
27435 /
27436 { - }
27437 \__fp_div_npos_o:Nww
27438 {
27439   \or:
27440   \__fp_case_use:nw
27441   { \__fp_division_by_zero_o:NNww \c_inf_fp / }
27442   \or:
27443   \__fp_case_use:nw
27444   { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
27445 }
27446 }

```

(End of definition for __fp/_o:ww.)

```

\__fp_div_npos_o:Nww \__fp_div_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign_A> {\exp
A}& \{A_1\} \{A_2\} \{A_3\} \{A_4\} \__fp_sep: \s__fp \__fp_chk:w 1
<sign_Z> {\exp Z}& \{Z_1\} \{Z_2\} \{Z_3\} \{Z_4\} \__fp_sep:

```

We want to compute A/Z . As for multiplication, `\__fp_sanitize:Nw` checks for overflow or underflow; we provide it with the `<final sign>`, and an integer expression in which we compute the exponent. We set up the arguments of `\__fp_div_significand_i_o:wnnw`, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{A_i\}$, then the four $\{Z_i\}$, a `\__fp_sep:`, and the `<final sign>`, used for rounding at the end.

```

27447 \cs_new:Npn \__fp_div_npos_o:Nww
27448 #1 \s__fp \__fp_chk:w 1 #2 #3 #4 \__fp_sep:
27449 \s__fp \__fp_chk:w 1 #5 #6 #7#8#9\__fp_sep:
27450 {
27451   \exp_after:wN \__fp_sanitize:Nw
27452   \exp_after:wN #1
27453   \int_value:w \__fp_int_eval:w
27454   #3 - #6
27455   \exp_after:wN \__fp_div_significand_i_o:wnnw
27456   \int_value:w \__fp_int_eval:w #7 \use_i:nnnn #8 + 1 \__fp_sep:
27457   #4
27458   {\#7}{\#8}\#9 \__fp_sep:
27459   #1
27460 }

```

(End of definition for __fp_div_npos_o:Nww.)

79.3.2 Work plan

In this subsection, we explain how to avoid overflowing $\text{T}_{\text{E}}\text{X}$'s integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.

- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\backslash\text{int\_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-TeX}$'s `\__fp_int_eval:w` rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int\_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned}
10^5 B &= A_1 A_2 0 + 10 \cdot 0 \cdot A_3 A_4 - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 \cdot Q_A \\
&< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1 \cdot Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 + 10 \\
&\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\
&\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y.
\end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned}
10^5 B &< 10^9 A/y + 1.6y, \\
10^5 C &< 10^9 B/y + 1.6y, \\
10^5 D &< 10^9 C/y + 1.6y, \\
10^5 E &< 10^9 D/y + 1.6y.
\end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned}
10^5 B &< 10^9/y + 1.6y, \\
10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\
10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\
10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2).
\end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned}
10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\
10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\
10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\
10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5).
\end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within $\text{T}_{\text{E}}\text{X}$'s bounds in all cases!

We later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε -TeX round

$$P = \text{\texttt{\textbackslash int\_eval:n}} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from ε -TeX’s rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

79.3.3 Implementing the significand division

`\_fp\_div\_significand\_i\_o:wnnw`

`\_fp\_div\_significand\_i\_o:wnnw <y> \_fp\_sep: {<A1>} {<A2>} {<A3>} {<A4>} {<Z1>} {<Z2>} {<Z3>} {<Z4>} \_fp\_sep: <sign>`

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `\_fp\_div\_significand\_calc:wnnnnnnnn`. Each of these calls needs $\langle y \rangle$ (#1), and it turns out that we need post-expansion there, hence the `\int\_value:w`. Here, #4 is six brace groups, which give the six first n-type arguments of the `calc` function.

```

27461 \cs_new:Npn \_fp\_div\_significand\_i\_o:wnnw #1 \_fp\_sep: #2#3 #4 \_fp\_sep:
27462 {
27463   \exp\_after:wN \_fp\_div\_significand\_test\_o:w
27464   \int\_value:w \_fp\_int\_eval:w
27465   \exp\_after:wN \_fp\_div\_significand\_calc:wnnnnnnnn
27466   \int\_value:w \_fp\_int\_eval:w 999999 + #2 #3 0 / #1 \_fp\_sep:
27467   #2 #3 \_fp\_sep:
27468   #4
27469   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
27470   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
27471   { \exp\_after:wN \_fp\_div\_significand\_ii:wN \int\_value:w #1 }
27472   { \exp\_after:wN \_fp\_div\_significand\_iii:wnnnnnn \int\_value:w #1 }
27473 }
```

(End of definition for `\_fp\_div\_significand\_i\_o:wnnw`.)

`\_fp\_div\_significand\_calc:wnnnnnnnn`
`\_fp\_div\_significand\_calc\_i:wnnnnnnnn`
`\_fp\_div\_significand\_calc\_ii:wnnnnnnnn`

`\_fp\_div\_significand\_calc:wnnnnnnnn <106 + QA> \_fp\_sep: <A1> <A2>`
`\_fp\_sep: {<A3>} {<A4>} {<Z1>} {<Z2>} {<Z3>} {<Z4>} {<continuation>}`
 expands to

$$\langle 10^6 + Q_A \rangle \langle continuation \rangle \backslash\_fp\_sep: \langle B_1 \rangle \langle B_2 \rangle \backslash\_fp\_sep: \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \\ \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \}$$

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_{\text{E}}\text{X}$'s bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a `\langle continuation \rangle`, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worse $-8 \cdot 10^8$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with $\text{T}_{\text{E}}\text{X}$'s limits once more.

```

27474 \cs_new:Npn \_fp_div_significand_calc:wwnnnnnnn #1
27475 {
27476   \if_meaning:w 1 #1
27477     \exp_after:wN \_fp_div_significand_calc_i:wwnnnnnnn
27478   \else:
27479     \exp_after:wN \_fp_div_significand_calc_ii:wwnnnnnnn
27480   \fi:
27481 }
27482 \cs_new:Npn \_fp_div_significand_calc_i:wwnnnnnnn
27483 #1\_fp_sep: #2\_fp_sep:#3#4 #5#6#7#8 #9
27484 {
27485   1 1 #1
27486   #9 \exp_after:wN \_fp_sep:
27487   \int_value:w \_fp_int_eval:w \c\_fp_Bigg_leading_shift_int
27488     + #2 - #1 * #5 - #5#60
27489   \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27490   \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
27491     + #3 - #1 * #6 - #70
27492   \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27493   \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
27494     + #4 - #1 * #7 - #80
27495   \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27496   \int_value:w \_fp_int_eval:w \c\_fp_Bigg_trailing_shift_int

```

```

27497         - #1 * #8 \_fp_sep:
27498         {#5}{#6}{#7}{#8}
27499     }
27500 \cs_new:Npn \_fp_div_significand_calc_ii:wnnnnnnn
27501 #1\_fp_sep: #2\_fp_sep:#3#4 #5#6#7#8 #9
27502 {
27503     1 0 #1
27504     #9 \exp_after:wN \_fp_sep:
27505     \int_value:w \_fp_int_eval:w \c\_fp_Bigg_leading_shift_int
27506     + #2 - #1 * #5
27507     \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27508     \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
27509     + #3 - #1 * #6
27510     \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27511     \int_value:w \_fp_int_eval:w \c\_fp_Bigg_middle_shift_int
27512     + #4 - #1 * #7
27513     \exp_after:wN \_fp_pack_Bigg:NNNNNNw
27514     \int_value:w \_fp_int_eval:w \c\_fp_Bigg_trailing_shift_int
27515     - #1 * #8 \_fp_sep:
27516     {#5}{#6}{#7}{#8}
27517 }

```

(End of definition for _fp_div_significand_calc:wnnnnnnn, _fp_div_significand_calc_i:wnnnnnnn, and _fp_div_significand_calc_ii:wnnnnnnn.)

_fp_div_significand_ii:wnn _fp_div_significand_ii:wnn $\langle y \rangle$ _fp_sep: $\langle B_1 \rangle$ _fp_sep: $\{\langle B_2 \rangle\}$ $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\langle continuations \rangle$ $\langle sign \rangle$
 Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result is output to the left, in an _fp_int_eval:w which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

27518 \cs_new:Npn \_fp_div_significand_ii:wnn #1\_fp_sep: #2\_fp_sep:#3
27519 {
27520     \exp_after:wN \_fp_div_significand_pack:NNN
27521     \int_value:w \_fp_int_eval:w
27522     \exp_after:wN \_fp_div_significand_calc:wnnnnnnn
27523     \int_value:w \_fp_int_eval:w
27524     999999 + #2 #3 0 / #1 \_fp_sep: #2 #3 \_fp_sep:
27525 }

```

(End of definition for _fp_div_significand_ii:wnn.)

_fp_div_significand_iii:wnnnnn _fp_div_significand_iii:wnnnnn $\langle y \rangle$ _fp_sep: $\langle E_1 \rangle$ _fp_sep: $\{\langle E_2 \rangle\}$ $\{\langle E_3 \rangle\}$ $\{\langle E_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\langle sign \rangle$
 We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D , the appropriate correction from a hypothetical Q_E .

```

27526 \cs_new:Npn \_fp_div_significand_iii:wnnnnn #1\_fp_sep: #2\_fp_sep:#3#4#5 #6#7
27527 {
27528     0
27529     \exp_after:wN \_fp_div_significand_iv:wnnnnnnn
27530     \int_value:w \_fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 \_fp_sep: % <- P
27531     #2 \_fp_sep: {#3} {#4} {#5}

```

```

27532     {#6} {#7}
27533 }

```

(End of definition for `\_fp_div_significand_iii:wwnnnnnn`.)

```

\_fp_div_significand_iv:wwnnnnnnn
\_fp_div_significand_v:NNw
\_fp_div_significand_vi:Nw

```

```

\_fp_div_significand_iv:wwnnnnnnn <P> \_fp_sep: <E1> \_fp_sep:
{<E2>} {<E3>} {<E4>} {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, i.e., if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of `\_fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

27534 \cs_new:Npn \_fp_div_significand_iv:wwnnnnnnn
27535   #1\_fp_sep: #2\_fp_sep:#3#4#5 #6#7#8#9
27536   {
27537     + 5 * #1
27538     \exp_after:wN \_fp_div_significand_vi:Nw
27539     \int_value:w \_fp_int_eval:w -50 + 2*#2#3 - #1*#6#7 +
27540     \exp_after:wN \_fp_div_significand_v:NN
27541     \int_value:w \_fp_int_eval:w 499950 + 2*#4 - #1*#8 +
27542     \exp_after:wN \_fp_div_significand_v:NN
27543     \int_value:w \_fp_int_eval:w 500000 + 2*#5 - #1*#9 \_fp_sep:
27544   }
27545 \cs_new:Npn \_fp_div_significand_v:NN #1#2 { #1#2 \_fp_int_eval_end: + }
27546 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2\_fp_sep:
27547   {
27548     \if_meaning:w 0 #1
27549     \if_int_compare:w \_fp_int_eval:w #2 > 0 + 1 \fi:
27550     \else:
27551     \if_meaning:w - #1 - \else: + \fi: 1
27552     \fi:
27553     \_fp_sep:
27554   }

```

(End of definition for `\__fp_div_significand_iv:wwnnnnnnn`, `\__fp_div_significand_v:NNw`, and `\__fp_div_significand_vi:Nw`.)

`\__fp_div_significand_pack:NNN` At this stage, we are in the following situation: $\text{T}_{\text{E}}\text{X}$ is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```
\__fp_div_significand_test_o:w 10^6 + Q_A \__fp_div_significand_-
pack:NNN 10^6 + Q_B \__fp_div_significand_pack:NNN 10^6 + Q_C \__fp_-
div_significand_pack:NNN 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon \__fp_sep: \langle sign \rangle
```

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, i.e., P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```
27555 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 \__fp_sep: }
```

(End of definition for `\__fp_div_significand_pack:NNN`.)

```
\__fp_div_significand_test_o:w \__fp_div_significand_test_o:w 10 \langle 5d \rangle \__fp_sep: \langle 4d \rangle \__fp_sep:
\langle 4d \rangle \__fp_sep: \langle 5d \rangle \__fp_sep: \langle sign \rangle
```

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence $\widetilde{Q}_A$ (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```
27556 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
27557 {
27558   \if_meaning:w 0 #1
27559     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
27560   \else:
27561     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
27562   \fi:
27563   #1
27564 }
```

(End of definition for `\__fp_div_significand_test_o:w`.)

```
\__fp_div_significand_small_o:wwwNNNNwN \__fp_div_significand_small_o:wwwNNNNwN 0 \langle 4d \rangle \__fp_sep: \langle 4d \rangle
\__fp_sep: \langle 4d \rangle \__fp_sep: \langle 5d \rangle \__fp_sep: \langle final sign \rangle
```

Standard use of the functions `\__fp_basics_pack_low:NNNNNw` and `\__fp_basics_pack_high:NNNNNw`. We finally get to use the $\langle final\ sign \rangle$ which has been sitting there for a while.

```
27565 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
27566   0 #1 \__fp_sep: #2 \__fp_sep: #3 \__fp_sep: #4 #5 #6 #7 #8 \__fp_sep: #9
27567 {
27568   \exp_after:wN \__fp_basics_pack_high:NNNNNw
27569   \int_value:w \__fp_int_eval:w 1 #1 #2
27570   \exp_after:wN \__fp_basics_pack_low:NNNNNw
27571   \int_value:w \__fp_int_eval:w 1 #3 #4 #5 #6 #7
27572   + \__fp_round:NNN #9 #7 #8
27573   \exp_after:wN \__fp_sep:
27574 }
```

(End of definition for `\__fp_div_significand_small_o:wwwNNNNwN`.)

_fp_div_significand_large_o:wwwNNNNwN

_fp_div_significand_large_o:wwwNNNNwN <5d> _fp_sep: <4d> _fp_sep: <4d> _fp_sep: <5d> _fp_sep: <sign>

We know that the final result cannot reach 10, hence 1#1#2, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the *<rounding digit>* from the last two of our 18 digits.

```

27575 \cs_new:Npn \_fp_div_significand_large_o:wwwNNNNwN
27576   #1\_fp_sep: #2\_fp_sep: #3\_fp_sep: #4#5#6#7#8\_fp_sep: #9
27577   {
27578     + 1
27579     \exp_after:wN \_fp_basics_pack_weird_high:NNNNNNNNw
27580     \int_value:w \_fp_int_eval:w 1 #1 #2
27581     \exp_after:wN \_fp_basics_pack_weird_low:NNNNw
27582     \int_value:w \_fp_int_eval:w 1 #3 #4 #5 #6 +
27583     \exp_after:wN \_fp_round:NNN
27584     \exp_after:wN #9
27585     \exp_after:wN #6
27586     \int_value:w \_fp_round_digit:Nw #7 #8 \_fp_sep:
27587     \exp_after:wN \_fp_sep:
27588   }

```

(End of definition for _fp_div_significand_large_o:wwwNNNNwN.)

79.4 Square root

_fp_sqrt_o:w Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and nan, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

27589 \cs_new:Npn \_fp_sqrt_o:w #1 \s_fp \_fp_chk:w #2#3#4\_fp_sep: @
27590   {
27591     \if_meaning:w 0 #2 \_fp_case_return_same_o:w \fi:
27592     \if_meaning:w 2 #3
27593       \_fp_case_use:nw { \_fp_invalid_operation_o:nw { sqrt } }
27594     \fi:
27595     \if_meaning:w 1 #2 \else: \_fp_case_return_same_o:w \fi:
27596     \_fp_sqrt_npos_o:w
27597     \s_fp \_fp_chk:w #2 #3 #4\_fp_sep:
27598   }

```

(End of definition for _fp_sqrt_o:w.)

_fp_sqrt_npos_o:w

Prepare _fp_sanitize:Nw to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

27599 \cs_new:Npn \_fp_sqrt_npos_o:w \s_fp \_fp_chk:w 1 0 #1#2#3#4#5\_fp_sep:
27600   {
27601     \exp_after:wN \_fp_sanitize:Nw
27602     \exp_after:wN 0
27603     \int_value:w \_fp_int_eval:w
27604     \if_int_odd:w #1 \exp_stop_f:

```

```

27605      \exp_after:wN \_fp_sqrt_npos_auxi_o:wwnnN
27606      \fi:
27607      #1 / 2
27608      \_fp_sqrt_Newton_o:wnn 56234133\_fp_sep: 0\_fp_sep: {#2#3} {#4#5} 0
27609    }
27610 \cs_new:Npn \_fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2\_fp_sep: 0\_fp_sep: #3#4#5
27611 {
27612   ( #1 + 1 ) / 2
27613   \_fp_pack_eight:wnnnnnnnN
27614   \_fp_sqrt_npos_auxii_o:wnnnnnnnN
27615   \_fp_sep:
27616   0 #3 #4
27617 }
27618 \cs_new:Npn \_fp_sqrt_npos_auxii_o:wnnnnnnnN #1\_fp_sep: #2#3#4#5#6#7#8#9
27619 { \_fp_sqrt_Newton_o:wnn 17782794\_fp_sep: 0\_fp_sep: {#1} {#2#3#4#5#6#7#8#9} }

```

(End of definition for $\_fp\_sqrt\_npos\_o:w$, $\_fp\_sqrt\_npos\_auxi\_o:wwnnN$, and $\_fp\_sqrt\_npos\_auxii\_o:wnnnnnnnN$.)

$\_fp\_sqrt\_Newton\_o:wnn$ Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes $\varepsilon\text{-TeX}$'s division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with $\varepsilon\text{-TeX}$ integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic-geometric inequality $(x + t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 1.125 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence $10^8 a_1/(x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges

to a single integer x . To stop the iteration when two consecutive results are equal, the function `\_fp_sqrt_Newton_o:wnn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

27620 \cs_new:Npn \_fp_sqrt_Newton_o:wnn #1\_fp_sep: #2\_fp_sep: #3
27621 {
27622   \if_int_compare:w #1 = #2 \exp_stop_f:
27623     \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnnN
27624     \int_value:w \_fp_int_eval:w 9999 9999 +
27625     \exp_after:wN \_fp_use_none_until_s:w
27626   \fi:
27627   \exp_after:wN \_fp_sqrt_Newton_o:wnn
27628   \int_value:w \_fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 \_fp_sep:
27629   #1\_fp_sep: {#3}
27630 }

```

(End of definition for `\_fp_sqrt_Newton_o:wnn`.)

`\_fp_sqrt_auxi_o:NNNNwnnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then `\_fp_sep: {⟨ $a_1$ ⟩} {⟨ $a_2$ ⟩} { $a'$ }`. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `\_fp_sqrt_auxii_o:NnnnnnnnnN` is called several times to get closer and closer underestimates of $\sqrt{a}$. By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

27631 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5\_fp_sep:
27632 {
27633   \_fp_sqrt_auxii_o:NnnnnnnnnN
27634   \_fp_sqrt_auxiii_o:wnnnnnnnnn
27635   {#1#2#3#4} {#5} {2499} {9988} {7500}
27636 }

```

(End of definition for `\_fp_sqrt_auxi_o:NNNNwnnnN`.)

`\_fp_sqrt_auxii_o:NnnnnnnnnN` This receives a continuation function `#1`, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[([10^{4j}(a - y^2)] - 257) \cdot (0.5 \cdot 10^8) / [10^8 y + 1] \right].$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a}-y$. On the one hand, $\sqrt{a}-y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a}-y \leq 5(\sqrt{a}+y)(\sqrt{a}-y) = 5(a-y^2) < 10^{9-4j}$. For $j = 3$, the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a}-y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a}-y) = \frac{10^{4j}(a-y^2-(\sqrt{a}-y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a-y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that ε -TeX's integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a}-y)$, hence $y+z \leq \sqrt{a}$ is an underestimate of $\sqrt{a}$, as claimed. One implementation detail: because the computation involves $-4*4 - 2*3*5 - 2*2*6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the `big` shifts.

```

27637 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
27638 {
27639   \exp_after:wN #1
27640   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
27641     + #7 - #2 * #2
27642   \exp_after:wN \__fp_pack_big:NNNNNNw
27643   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27644     - 2 * #2 * #3
27645   \exp_after:wN \__fp_pack_big:NNNNNNw
27646   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27647     + #8 - #3 * #3 - 2 * #2 * #4
27648   \exp_after:wN \__fp_pack_big:NNNNNNw
27649   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27650     - 2 * #3 * #4 - 2 * #2 * #5
27651   \exp_after:wN \__fp_pack_big:NNNNNNw
27652   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27653     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
27654   \exp_after:wN \__fp_pack_big:NNNNNNw
27655   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27656     - 2 * #4 * #5 - 2 * #3 * #6
27657   \exp_after:wN \__fp_pack_big:NNNNNNw
27658   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
27659     - #5 * #5 - 2 * #4 * #6
27660   \exp_after:wN \__fp_pack_big:NNNNNNw
27661   \int_value:w \__fp_int_eval:w
27662     \c__fp_big_middle_shift_int
27663     - 2 * #5 * #6
27664   \exp_after:wN \__fp_pack_big:NNNNNNw
27665   \int_value:w \__fp_int_eval:w
27666     \c__fp_big_trailing_shift_int
27667     - #6 * #6 \__fp_sep:
27668   % (
27669     - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 \__fp_sep:
27670     {#2}{#3}{#4}{#5}{#6} {#7}{#8}{#9}
27671 }

```

(End of definition for `\__fp_sqrt_auxii_o:NnnnnnnN`.)


```

\__fp_sqrt_auxiii_o:wnnnnnnnn
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle \setminus \_\_\text{fp\_sep}: \{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller $\setminus \_\_\text{fp\_sqrt\_auxii\_o:NnnnnnnnN}$, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) \right] / \lfloor 10^8 y + 1 \rfloor$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j = 4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8d_3 + 10^4d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded $\sqrt{a}$ with only one more call to $\setminus \_\_\text{fp\_sqrt\_auxii\_o:NnnnnnnnN}$. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8 y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

27672 \cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
27673   #1\__fp_sep: #2#3#4#5#6#7#8#9
27674   {
27675     \if_int_compare:w #1 > \c_one_int
27676       \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
27677       \int_value:w \__fp_int_eval:w (#1#2 %)
27678     \else:
27679       \if_int_compare:w #1#2 > \c_one_int
27680       \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
27681       \int_value:w \__fp_int_eval:w (#1#2#3 %)
27682     \else:
27683       \if_int_compare:w #1#2#3 > \c_one_int
27684       \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
27685       \int_value:w \__fp_int_eval:w (#1#2#3#4 %)
27686     \else:
27687       \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
27688       \int_value:w \__fp_int_eval:w (#1#2#3#4#5 %)
27689     \fi:
27690   \fi:
27691 \fi:
27692 }
27693 \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6\__fp_sep:
27694   { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
27695 \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6\__fp_sep:
27696   { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
27697 \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6\__fp_sep:
27698   { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
27699 \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6\__fp_sep:
27700   {
27701     \if_int_compare:w #1#2 = \c_zero_int
27702     \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
27703   \fi:

```

```

27704     \_fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
27705 }

```

(End of definition for _fp_sqrt_auxiii_o:wnnnnnnn and others.)

_fp_sqrt_auxviii_o:nnnnnnnn Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks of y , then call the auxii auxiliary to evaluate $y'^2 = (y + z)^2$.

```

27706 \cs_new:Npn \_fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
27707 {
27708   \exp_after:wN \_fp_sqrt_auxix_o:wnwnw
27709   \int_value:w \_fp_int_eval:w #3
27710   \exp_after:wN \_fp_basics_pack_low:NNNNNw
27711   \int_value:w \_fp_int_eval:w #1 + 1#4#5
27712   \exp_after:wN \_fp_basics_pack_low:NNNNNw
27713   \int_value:w \_fp_int_eval:w #2 + 1#6#7 \_fp_sep:
27714 }
27715 \cs_new:Npn \_fp_sqrt_auxix_o:wnwnw #1\_fp_sep: #2#3\_fp_sep: #4#5\_fp_sep:
27716 {
27717   \_fp_sqrt_auxii_o:NnnnnnnnnN
27718   \_fp_sqrt_auxiii_o:wnnnnnnnnn {#1}{#2}{#3}{#4}{#5}
27719 }

```

(End of definition for _fp_sqrt_auxviii_o:nnnnnnnn and _fp_sqrt_auxix_o:wnwnw.)

_fp_sqrt_auxx_o:Nnnnnnnnn At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
 _fp_sqrt_auxxi_o:wnnnN

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of $\sqrt{a}$ and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits #8 of y are considered, and we do not perform any carry yet. The auxxi auxiliary sets up auxii with a continuation function auxxii instead of auxiii as before. To prevent auxii from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

27720 \cs_new:Npn \_fp_sqrt_auxx_o:Nnnnnnnnn #1#2#3 #4#5#6#7#8
27721 {
27722   \exp_after:wN \_fp_sqrt_auxxi_o:wnnnN
27723   \int_value:w \_fp_int_eval:w
27724   (#8 + 2499) / 5000 * 5000 \_fp_sep:
27725   {#4} {#5} {#6} {#7} \_fp_sep:
27726 }
27727 \cs_new:Npn \_fp_sqrt_auxxi_o:wnnnN #1\_fp_sep: #2\_fp_sep: #3#4#5
27728 {
27729   \_fp_sqrt_auxii_o:NnnnnnnnnN

```

```

27730      \__fp_sqrt_auxxii_o:nnnnnnnnnw
27731      #2 {#1}
27732      {#3} { #4 + 1 } #5
27733  }

```

(End of definition for __fp_sqrt_auxx_o:Nnnnnnnn and __fp_sqrt_auxxi_o:wnnnN.)

__fp_sqrt_auxxii_o:nnnnnnnnnw
 __fp_sqrt_auxxiii_o:w

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the auxxiv function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

27734 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnnw 0\__fp_sep: #1#2#3#4#5#6#7#8 #9\__fp_sep:
27735 {
27736   \if_int_compare:w #1#2 > \c_zero_int
27737     \if_int_compare:w #1#2 = \c_one_int
27738       \if_int_compare:w #3#4 = \c_zero_int
27739         \if_int_compare:w #5#6 = \c_zero_int
27740           \if_int_compare:w #7#8 = \c_zero_int
27741             \__fp_sqrt_auxxiii_o:w
27742           \fi:
27743         \fi:
27744       \fi:
27745     \fi:
27746     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnnN
27747     \int_value:w 9998
27748   \else:
27749     \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnnN
27750     \int_value:w 10000
27751   \fi:
27752   \__fp_sep:
27753 }
27754 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: \__fp_sep:
27755 {
27756   \fi: \fi: \fi: \fi: \fi:
27757   \__fp_sqrt_auxxiv_o:wnnnnnnnnN 9999 \__fp_sep:
27758 }

```

(End of definition for __fp_sqrt_auxxii_o:nnnnnnnnnw and __fp_sqrt_auxxiii_o:w.)

__fp_sqrt_auxxiv_o:wnnnnnnnnN

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by __fp_round:NNN, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by __fp_round_digit:Nw, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

27759 \cs_new:Npn \__fp_sqrt_auxxiv_o:w nnnnnnnN #1\__fp_sep: #2#3#4#5#6 #7#8#9
27760 {
27761   \exp_after:wN \__fp_basics_pack_high:NNNNNw
27762   \int_value:w \__fp_int_eval:w 1 0000 0000 + #2#3
27763   \exp_after:wN \__fp_basics_pack_low:NNNNNw
27764   \int_value:w \__fp_int_eval:w 1 0000 0000
27765   + #4#5
27766   \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
27767   + \exp_after:wN \__fp_round:NNN
27768   \exp_after:wN 0
27769   \exp_after:wN 0
27770   \int_value:w
27771   \exp_after:wN \use_i:nn
27772   \exp_after:wN \__fp_round_digit:Nw
27773   \int_value:w \__fp_int_eval:w #6 + 19999 - #1 \__fp_sep:
27774   \exp_after:wN \__fp_sep:
27775 }

```

(End of definition for __fp_sqrt_auxxiv_o:w nnnnnnnN.)

79.5 About the sign and exponent

__fp_logb_o:w The exponent of a normal number is its *⟨exponent⟩* minus one.

```

\__fp_logb_aux_o:w 27776 \cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2\__fp_sep: @
27777 {
27778   \if_case:w #1 \exp_stop_f:
27779   \__fp_case_use:nw
27780   { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
27781   \or: \exp_after:wN \__fp_logb_aux_o:w
27782   \or: \__fp_case_return_o:Nw \c_inf_fp
27783   \else: \__fp_case_return_same_o:w
27784   \fi:
27785   \s__fp \__fp_chk:w #1 #2\__fp_sep:
27786 }
27787 \cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 \__fp_sep:
27788 {
27789   \exp_after:wN \__fp_parse:n \exp_after:wN
27790   { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
27791 }

```

(End of definition for __fp_logb_o:w and __fp_logb_aux_o:w.)

__fp_sign_o:w Find the sign of the floating point: nan, +0, -0, +1 or -1.

```

\__fp_sign_aux_o:w 27792 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2\__fp_sep: @
27793 {
27794   \if_case:w #1 \exp_stop_f:
27795   \__fp_case_return_same_o:w
27796   \or: \exp_after:wN \__fp_sign_aux_o:w
27797   \or: \exp_after:wN \__fp_sign_aux_o:w
27798   \else: \__fp_case_return_same_o:w
27799   \fi:
27800   \s__fp \__fp_chk:w #1 #2\__fp_sep:
27801 }

```

```

27802 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 \__fp_sep:
27803 { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End of definition for __fp_sign_o:w and __fp_sign_aux_o:w.)

__fp_set_sign_o:w This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like __fp+_o:ww.

```

27804 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
27805 {
27806   \exp_after:wN \__fp_exp_after_o:w
27807   \exp_after:wN \s__fp
27808   \exp_after:wN \__fp_chk:w
27809   \exp_after:wN #2
27810   \int_value:w
27811   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
27812   #4\__fp_sep:
27813 }

```

(End of definition for __fp_set_sign_o:w.)

79.6 Operations on tuples

__fp_tuple_set_sign_o:w Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

__fp_tuple_set_sign_aux_o:w

```

27814 \cs_new:Npn \__fp_tuple_set_sign_o:w #1#2 @
27815 {
27816   \if_meaning:w 2 #1
27817   \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw
27818   \fi:
27819   \__fp_invalid_operation_o:nw { abs }
27820   #2
27821 }
27822 \cs_new:Npn \__fp_tuple_set_sign_aux_o:Nnw #1#2
27823 { \__fp_tuple_map_o:nw \__fp_tuple_set_sign_aux_o:w }
27824 \cs_new:Npn \__fp_tuple_set_sign_aux_o:w #1#2 \__fp_sep:
27825 {
27826   \__fp_change_func_type:NNN #1 \__fp_set_sign_o:w
27827   \__fp_parse_apply_unary_error:NNw
27828   2 #1 #2 \__fp_sep: @
27829 }

```

(End of definition for __fp_tuple_set_sign_o:w, __fp_tuple_set_sign_aux_o:Nnw, and __fp_tuple_set_sign_aux_o:w.)

__fp*_tuple_o:ww For `<number>*<tuple>` and `<tuple>*<number>` and `<tuple>/<number>`, loop through the __fp_tuple*_o:ww `<tuple>` some code that multiplies or divides by the appropriate `<number>`. Importantly __fp_tuple/_o:ww we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```

27830 \cs_new:cpn { \__fp*_tuple_o:ww } #1 \__fp_sep:
27831 { \__fp_tuple_map_o:nw { \__fp_binary_type_o:Nnw * #1 \__fp_sep: } }

```

```

27832 \cs_new:cpn { __fp_tuple*_o:ww } #1 \__fp_sep: #2 \__fp_sep:
27833 {
27834   \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww * #2 \__fp_sep: }
27835   #1 \__fp_sep:
27836 }
27837 \cs_new:cpn { __fp_tuple/_o:ww } #1 \__fp_sep: #2 \__fp_sep:
27838 {
27839   \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww / #2 \__fp_sep: }
27840   #1 \__fp_sep:
27841 }

```

(End of definition for __fp_*_tuple_o:ww, __fp_tuple*_o:ww, and __fp_tuple/_o:ww.)

__fp_tuple+_tuple_o:ww Check the two tuples have the same number of items and map through these a helper
 __fp_tuple-_tuple_o:ww that dispatches appropriately depending on the types. This means (1,2)+((1,1),2)
 gives (nan,4).

```

27842 \cs_set_protected:Npn \__fp_tmp:w #1
27843 {
27844   \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
27845   \s__fp_tuple \__fp_tuple_chk:w ##1 \__fp_sep:
27846   \s__fp_tuple \__fp_tuple_chk:w ##2 \__fp_sep:
27847   {
27848     \int_compare:nNnTF
27849     { \__fp_array_count:n {##1} } = { \__fp_array_count:n {##2} }
27850     { \__fp_tuple_mapthread_o:nww { \__fp_binary_type_o:Nww #1 } }
27851     { \__fp_invalid_operation_o:nww #1 }
27852     \s__fp_tuple \__fp_tuple_chk:w {##1} \__fp_sep:
27853     \s__fp_tuple \__fp_tuple_chk:w {##2} \__fp_sep:
27854   }
27855 }
27856 \__fp_tmp:w +
27857 \__fp_tmp:w -

```

(End of definition for __fp_tuple+_tuple_o:ww and __fp_tuple-_tuple_o:ww.)

27858 </code>

Chapter 80

l3fp-extended implementation

27859 $\langle *code \rangle$

27860 $\langle @@=fp \rangle$

80.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{ \langle a_1 \rangle \} \{ \langle a_2 \rangle \} \{ \langle a_3 \rangle \} \{ \langle a_4 \rangle \} \{ \langle a_5 \rangle \} \{ \langle a_6 \rangle \} \backslash\_fp\_sep:$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

$$\backslash\_fp\_fixed\_ \langle calculation \rangle : wwn \langle operand_1 \rangle \backslash\_fp\_sep: \\ \langle operand_2 \rangle \backslash\_fp\_sep: \{ \langle continuation \rangle \}$$

They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a $\backslash\_fp\_sep:$) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

$$\backslash\_fp\_fixed\_add: wwn \langle X_1 \rangle \backslash\_fp\_sep: \langle X_2 \rangle \backslash\_fp\_sep: \\ \backslash\_fp\_fixed\_mul: wwn \langle X_3 \rangle \backslash\_fp\_sep: \\ \backslash\_fp\_fixed\_add: wwn \langle X_4 \rangle \backslash\_fp\_sep:$$

to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `\__fp_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

80.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in `l3fp-expo`.

```
27861 \tl_const:Nn \c__fp_one_fixed_tl
27862 { {10000} {0000} {0000} {0000} {0000} {0000} \__fp_sep: }
```

(End of definition for `\c__fp_one_fixed_tl`.)

`\__fp_fixed_continue:wn` This function simply calls the next function.

```
27863 \cs_new:Npn \__fp_fixed_continue:wn #1\__fp_sep: #2 { #2 #1\__fp_sep: }
```

(End of definition for `\__fp_fixed_continue:wn`.)

`\__fp_fixed_add_one:wN` `\__fp_fixed_add_one:wN` $\langle a \rangle$ `\__fp_sep:` $\langle continuation \rangle$

This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 + 10000 < 2^{31}$.

```
27864 \cs_new:Npn \__fp_fixed_add_one:wN #1#2\__fp_sep: #3
27865 {
27866   \exp_after:wN #3 \exp_after:wN
27867   { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 \__fp_sep:
27868 }
```

(End of definition for `\__fp_fixed_add_one:wN`.)

`\__fp_fixed_div_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group #1 may have any number of digits, and we must split #1 into the new first group and a second group of exactly 4 digits. The choice of shifts allows #1 to be in the range $[0, 5 \cdot 10^8 - 1]$.

```
27869 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6\__fp_sep:
27870 {
27871   \exp_after:wN \__fp_fixed_mul_after:wnn
27872   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
27873   \exp_after:wN \__fp_pack:NNNNNw
27874   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
27875   + #1 \__fp_sep: {#2}{#3}{#4}{#5}\__fp_sep:
27876 }
```

(End of definition for `\__fp_fixed_div_myriad:wn`.)

`\__fp_fixed_mul_after:wnn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #3 in front.

```
27877 \cs_new:Npn \__fp_fixed_mul_after:wnn #1\__fp_sep: #2\__fp_sep: #3
27878 { #3 {#1} #2\__fp_sep: }
```

(End of definition for `\__fp_fixed_mul_after:wnn`.)

80.3 Multiplying a fixed point number by a short one

`\_fp_fixed_mul_short:wnn`

```
\_fp_fixed_mul_short:wnn
  {⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} \_fp_sep:
  {⟨b0⟩} {⟨b1⟩} {⟨b2⟩} \_fp_sep: {⟨continuation⟩}
```

Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \} \_fp\_sep:$ in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: for instance a second operand of $\{0001\}\{0000\}\{0000\}$ leaves the first operand unchanged (rather than dividing it by 10^4 , as `\_fp_fixed_mul:wnn` would).

```
27879 \cs_new:Npn \_fp_fixed_mul_short:wnn #1#2#3#4#5#6\_fp_sep: #7#8#9\_fp_sep:
27880 {
27881   \exp_after:wN \_fp_fixed_mul_after:wnn
27882   \int_value:w \_fp_int_eval:w \c\_fp_leading_shift_int
27883     + #1*#7
27884   \exp_after:wN \_fp_pack:NNNNNw
27885   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
27886     + #1*#8 + #2*#7
27887   \exp_after:wN \_fp_pack:NNNNNw
27888   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
27889     + #1*#9 + #2*#8 + #3*#7
27890   \exp_after:wN \_fp_pack:NNNNNw
27891   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
27892     + #2*#9 + #3*#8 + #4*#7
27893   \exp_after:wN \_fp_pack:NNNNNw
27894   \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
27895     + #3*#9 + #4*#8 + #5*#7
27896   \exp_after:wN \_fp_pack:NNNNNw
27897   \int_value:w \_fp_int_eval:w \c\_fp_trailing_shift_int
27898     + #4*#9 + #5*#8 + #6*#7
27899     + ( #5*#9 + #6*#8 + #6*#9 / \c\_fp_myriad_int )
27900     / \c\_fp_myriad_int \_fp_sep: \_fp_sep:
27901 }
```

(End of definition for `\_fp_fixed_mul_short:wnn`.)

80.4 Dividing a fixed point number by a small integer

`\_fp_fixed_div_int:wnN`

`\_fp_fixed_div_int:wnN`

`\_fp_fixed_div_int_auxi:wnn`

`\_fp_fixed_div_int_auxii:wnn`

`\_fp_fixed_div_int_pack:Nw`

`\_fp_fixed_div_int_after:Nw`

```
\_fp_fixed_div_int:wnN ⟨a⟩ \_fp_sep: ⟨n⟩ \_fp_sep: ⟨continuation⟩
```

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

The arguments of the `i` auxiliary are 1: one of the a_i , 2: n , 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .

The `ii` auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1} . The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q1
\__fp_fixed_div_int_pack:Nw 9999 + Q2
\__fp_fixed_div_int_pack:Nw 9999 + Q3
\__fp_fixed_div_int_pack:Nw 9999 + Q4
\__fp_fixed_div_int_pack:Nw 9999 + Q5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q6 \__fp_sep: {<n>} {<a6>}

```

where expansion is happening from the last line up. The `iii` auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each `pack` auxiliary receives 5 digits followed by a `\__fp_sep:`. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the `<continuation>` as appropriate.

```

27902 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 \__fp_sep: #7 \__fp_sep: #8
27903 {
27904   \exp_after:wN \__fp_fixed_div_int_after:Nw
27905   \exp_after:wN #8
27906   \int_value:w \__fp_int_eval:w - 1
27907   \__fp_fixed_div_int:wnN
27908   #1\__fp_sep: {#7} \__fp_fixed_div_int_auxi:wnn
27909   #2\__fp_sep: {#7} \__fp_fixed_div_int_auxi:wnn
27910   #3\__fp_sep: {#7} \__fp_fixed_div_int_auxi:wnn
27911   #4\__fp_sep: {#7} \__fp_fixed_div_int_auxi:wnn
27912   #5\__fp_sep: {#7} \__fp_fixed_div_int_auxi:wnn
27913   #6\__fp_sep: {#7} \__fp_fixed_div_int_auxii:wnn \__fp_sep:
27914 }
27915 \cs_new:Npn \__fp_fixed_div_int:wnN #1\__fp_sep: #2 #3
27916 {
27917   \exp_after:wN #3
27918   \int_value:w \__fp_int_eval:w #1 / #2 - 1 \__fp_sep:
27919   {#2}
27920   {#1}
27921 }
27922 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1\__fp_sep: #2 #3
27923 {
27924   + #1
27925   \exp_after:wN \__fp_fixed_div_int_pack:Nw
27926   \int_value:w \__fp_int_eval:w 9999
27927   \exp_after:wN \__fp_fixed_div_int:wnN
27928   \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
27929 }
27930 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1\__fp_sep: #2 #3 { + #1 + 2 \__fp_sep: }
27931 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2\__fp_sep: { + #1\__fp_sep: {#2} }
27932 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2\__fp_sep: { #1 {#2} }

```

(End of definition for `\__fp_fixed_div_int:wwN` and others.)

80.5 Adding and subtracting fixed points

```

\__fp_fixed_add:wnn          \__fp_fixed_add:wnn <a> \__fp_sep: <b> \__fp_sep: {<continuation>}
\__fp_fixed_sub:wnn
\__fp_fixed_add:Nnnnnwnn
\__fp_fixed_add:nnNnnnwn
\__fp_fixed_add_pack:NNNNNwn
\__fp_fixed_add_after:NNNNNwn

```

Computes $a+b$ (resp. $a-b$) and feeds the result to the $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, $a_1, \dots, a_4$, the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$ (#8, then #7) from the end of the argument list to its start.

```

27933 \cs_new:Npn \__fp_fixed_add:wnn { \__fp_fixed_add:Nnnnnwnn + }
27934 \cs_new:Npn \__fp_fixed_sub:wnn { \__fp_fixed_add:Nnnnnwnn - }
27935 \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6\__fp_sep: #7#8
27936 {
27937   \exp_after:wN \__fp_fixed_add_after:NNNNNwn
27938   \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
27939   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
27940   \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
27941   \__fp_fixed_add:nnNnnwn #6 #1
27942 }
27943 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 \__fp_sep: #8
27944 {
27945   #3 #4#5
27946   \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
27947   \int_value:w \__fp_int_eval:w
27948   2 0000 0000 #3 #6#7 + #1#2 \__fp_sep: {#8} \__fp_sep:
27949 }
27950 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6\__fp_sep: #7
27951 { + #1 \__fp_sep: {#7} {#2#3#4#5} {#6} }
27952 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6\__fp_sep: #7
27953 { #7 {#1#2#3#4#5} {#6} }

```

(End of definition for $\__fp\_fixed\_add:wnn$ and others.)

80.6 Multiplying fixed points

$\__fp\_fixed\_mul:wnn$
 $\__fp\_fixed\_mul:nnnnnnnw$

$\__fp\_fixed\_mul:wnn \langle a \rangle \__fp\_sep: \langle b \rangle \__fp\_sep: \{\langle continuation \rangle\}$

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for $\text{\TeX}$ macros. Note that we don't need to obtain an exact rounding, contrarily to the $*$ operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on $a_1, \dots, a_4$ and $b_1, \dots, b_4$, while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs $a_1, \dots, a_4$, the rest of a , and $b_1, \dots, b_4$, and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `\__fp_fixed_mul_after:wwn`.

```

27954 \cs_new:Npn \__fp_fixed_mul:wwn #1#2#3#4 #5\__fp_sep: #6#7#8#9
27955 {
27956   \exp_after:wN \__fp_fixed_mul_after:wwn
27957   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
27958   \exp_after:wN \__fp_pack:NNNNNw
27959   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27960   + #1*#6
27961   \exp_after:wN \__fp_pack:NNNNNw
27962   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27963   + #1*#7 + #2*#6
27964   \exp_after:wN \__fp_pack:NNNNNw
27965   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27966   + #1*#8 + #2*#7 + #3*#6
27967   \exp_after:wN \__fp_pack:NNNNNw
27968   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27969   + #1*#9 + #2*#8 + #3*#7 + #4*#6
27970   \exp_after:wN \__fp_pack:NNNNNw
27971   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
27972   + #2*#9 + #3*#8 + #4*#7
27973   + ( #3*#9 + #4*#8
27974     + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
27975   )
27976 \cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 \__fp_sep:
27977 {
27978   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
27979   + #1*#3 + #5*#7 \__fp_sep: \__fp_sep:
27980 }

```

(End of definition for `\__fp_fixed_mul:wwn` and `\__fp_fixed_mul:nnnnnnnw`.)

80.7 Combining product and sum of fixed points

```

\__fp_fixed_mul_add:wwwn <a> \__fp_sep: <b> \__fp_sep: <c> \__fp_sep:
{<continuation>}
\__fp_fixed_mul_sub_back:wwwn <a> \__fp_sep: <b> \__fp_sep: <c> \__fp_sep:
{<continuation>}
\__fp_fixed_mul_one_minus_mul:wwwn <a> \__fp_sep: <b> \__fp_sep: {<continuation>}

```

Sometimes called FMA (fused multiply-add), these functions compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where $c_1 c_2, c_3 c_4, c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c , and $\cdot$ denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4} , is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing `+ c_5 c_6 \_\_\_fp\_sep: {\langle continuation \rangle} \_\_\_fp\_sep:`. The `+ c_5 c_6` piece, which is omitted for `\_\_\_fp\_fixed\_one\_minus\_mul:wwn`, is taken in the integer expression for the 10^{-24} level.

```

27981 \cs_new:Npn \_\_\_fp\_fixed\_mul\_add:wwn #1\_\_\_fp\_sep: #2\_\_\_fp\_sep: #3#4#5#6#7#8\_\_\_fp\_sep:
27982 {
27983   \exp\_after:wN \_\_\_fp\_fixed\_mul\_after:wwn
27984   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
27985   \exp\_after:wN \_\_\_fp\_pack\_big:NNNNNNw
27986   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
27987   \_\_\_fp\_fixed\_mul\_add:Nwnnnwnnn +
27988   + #5 #6 \_\_\_fp\_sep: #2 \_\_\_fp\_sep: #1 \_\_\_fp\_sep: #2 \_\_\_fp\_sep: +
27989   + #7 #8 \_\_\_fp\_sep: \_\_\_fp\_sep:
27990 }
27991 \cs_new:Npn \_\_\_fp\_fixed\_mul\_sub\_back:wwn
27992 #1\_\_\_fp\_sep: #2\_\_\_fp\_sep: #3#4#5#6#7#8\_\_\_fp\_sep:
27993 {
27994   \exp\_after:wN \_\_\_fp\_fixed\_mul\_after:wwn
27995   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
27996   \exp\_after:wN \_\_\_fp\_pack\_big:NNNNNNw
27997   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
27998   \_\_\_fp\_fixed\_mul\_add:Nwnnnwnnn -
27999   + #5 #6 \_\_\_fp\_sep: #2 \_\_\_fp\_sep: #1 \_\_\_fp\_sep: #2 \_\_\_fp\_sep: -
28000   + #7 #8 \_\_\_fp\_sep: \_\_\_fp\_sep:
28001 }
28002 \cs_new:Npn \_\_\_fp\_fixed\_one\_minus\_mul:wwn #1\_\_\_fp\_sep: #2\_\_\_fp\_sep:
28003 {
28004   \exp\_after:wN \_\_\_fp\_fixed\_mul\_after:wwn
28005   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
28006   \exp\_after:wN \_\_\_fp\_pack\_big:NNNNNNw
28007   \int\_value:w \_\_\_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int +
28008   1 0000 0000
28009   \_\_\_fp\_fixed\_mul\_add:Nwnnnwnnn -
28010   \_\_\_fp\_sep: #2 \_\_\_fp\_sep: #1 \_\_\_fp\_sep: #2 \_\_\_fp\_sep: -
28011   \_\_\_fp\_sep: \_\_\_fp\_sep:

```

28012 }

(End of definition for `\_fp\_fixed\_mul\_add:wwn`, `\_fp\_fixed\_mul\_sub\_back:wwn`, and `\_fp\_fixed\_mul\_one\_minus\_mul:wwn`.)

`\_fp\_fixed\_mul\_add:Nwnnnwnnn` $\langle op \rangle + \langle c_3 \rangle \langle c_4 \rangle \_fp\_sep:$
 $\langle b \rangle \_fp\_sep: \langle a \rangle \_fp\_sep: \langle b \rangle \_fp\_sep: \langle op \rangle$
 $+ \langle c_5 \rangle \langle c_6 \rangle \_fp\_sep:$

Here, $\langle op \rangle$ is either $+$ or $-$. Arguments #3, #4, #5 are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The a - b products use the sign #1. Note that #2 is empty for `\_fp\_fixed\_one\_minus\_mul:wwn`. We call the *ii* auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

28013 `\cs\_new:Npn \_fp\_fixed\_mul\_add:Nwnnnwnnn #1 #2\_fp\_sep: #3#4#5#6\_fp\_sep: #7#8#9`
28014 `{`
28015 `#1 #7*#3`
28016 `\exp\_after:wN \_fp\_pack\_big:NNNNNNw`
28017 `\int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int`
28018 `#1 #7*#4 #1 #8*#3`
28019 `\exp\_after:wN \_fp\_pack\_big:NNNNNNw`
28020 `\int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int`
28021 `#1 #7*#5 #1 #8*#4 #1 #9*#3 #2`
28022 `\exp\_after:wN \_fp\_pack\_big:NNNNNNw`
28023 `\int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int`
28024 `#1 \_fp\_fixed\_mul\_add:nnnnwnnn {#7}{#8}{#9}`
28025 `}`

(End of definition for `\_fp\_fixed\_mul\_add:Nwnnnwnnn`.)

`\_fp\_fixed\_mul\_add:nnnnwnnn` $\langle a \rangle \_fp\_sep: \langle b \rangle \_fp\_sep: \langle op \rangle$
 $+ \langle c_5 \rangle \langle c_6 \rangle \_fp\_sep:$

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1 , a_5 , a_6 , and the corresponding pieces of $\langle b \rangle$.

28026 `\cs\_new:Npn \_fp\_fixed\_mul\_add:nnnnwnnn #1#2#3#4#5\_fp\_sep: #6#7#8#9`
28027 `{`
28028 `( #1*#9 + #2*#8 + #3*#7 + #4*#6 )`
28029 `\exp\_after:wN \_fp\_pack\_big:NNNNNNw`
28030 `\int\_value:w \_fp\_int\_eval:w \c\_fp\_big\_trailing\_shift\_int`
28031 `\_fp\_fixed\_mul\_add:nnnnwnnn`
28032 `{ #6 + #4*#7 + #3*#8 + #2*#9 + #1 }`
28033 `{ #7 + #4*#8 + #3*#9 + #2 }`
28034 `{#1} #5\_fp\_sep:`
28035 `{#6}`
28036 `}`

(End of definition for `\_fp_fixed_mul_add:nnnnwnnnn`.)

```
\_fp_fixed_mul_add:nnnnwnnnN
    \_fp_fixed_mul_add:nnnnwnnnN {<partial1>} {<partial2>}
    {<a1>} {<a5>} {<a6>} \_fp_sep: {<b1>} {<b5>} {<b6>} \_fp_sep:
    <op> + <c5> <c6> \_fp_sep:
```

Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+c_5c_6$ is taken into the expression for level 10^{-24} . Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See `l3fp-aux` for the definition of the shifts and packing auxiliaries.

```
28037 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnN #1#2 #3#4#5\_fp_sep: #6#7#8\_fp_sep: #9
28038 {
28039     #9 (#4* #1 *#7)
28040     #9 (#5*#6+#4* #2 *#7+#3*#8) / \c__fp_myriad_int
28041 }
```

(End of definition for `\_fp_fixed_mul_add:nnnnwnnnn`.)

80.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a `\_fp_sep:`. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.

```
\_fp_ep_to_fixed:wwn
\_fp_ep_to_fixed_auxi:www
\_fp_ep_to_fixed_auxii:nnnnnnwn
```

Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```
28042 \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
28043 {
28044     \exp_after:wN \_fp_ep_to_fixed_auxi:www
28045     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN \_fp_sep:
28046     \exp:w \exp_end_continue_f:w
28047     \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } \_fp_sep:
28048 }
28049 \cs_new:Npn \_fp_ep_to_fixed_auxi:www
28050 1#1\_fp_sep: #2\_fp_sep: #3#4#5#6#7\_fp_sep:
28051 {
28052     \_fp_pack_eight:wNNNNNNNN
28053     \_fp_pack_twice_four:wNNNNNNNN
28054     \_fp_pack_twice_four:wNNNNNNNN
28055     \_fp_pack_twice_four:wNNNNNNNN
28056     \_fp_ep_to_fixed_auxii:nnnnnnwn \_fp_sep:
28057     #2 #1#3#4#5#6#7 0000 !
28058 }
28059 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7\_fp_sep: #8! #9
28060 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}\_fp_sep: }
```

(End of definition for `\__fp_ep_to_fixed:wwn`, `\__fp_ep_to_fixed_auxi:www`, and `\__fp_ep_to_fixed_auxii:nnnnnnwnn`.)

`\__fp_ep_to_ep:wwN`
`\__fp_ep_to_ep_loop:N`
`\__fp_ep_to_ep_end:www`
`\__fp_ep_to_ep_zero:ww`

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation `#8` is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `\__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

28061 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7\__fp_sep: #8
28062 {
28063   \exp_after:wN #8
28064   \int_value:w \__fp_int_eval:w #1 + 4
28065   \exp_after:wN \use_i:nn
28066   \exp_after:wN \__fp_ep_to_ep_loop:N
28067   \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
28068   #3#4#5#6#7 \__fp_sep: \__fp_sep: !
28069 }
28070 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
28071 {
28072   \if_meaning:w 0 #1
28073   - 1
28074   \else:
28075     \__fp_ep_to_ep_end:www #1
28076   \fi:
28077   \__fp_ep_to_ep_loop:N
28078 }
28079 \cs_new:Npn \__fp_ep_to_ep_end:www
28080 #1 \fi: \__fp_ep_to_ep_loop:N #2\__fp_sep: #3!
28081 {
28082   \fi:
28083   \if_meaning:w \__fp_sep: #1
28084   - 2 * \c__fp_max_exponent_int
28085   \__fp_ep_to_ep_zero:ww
28086   \fi:
28087   \__fp_pack_twice_four:wNNNNNNNN
28088   \__fp_pack_twice_four:wNNNNNNNN
28089   \__fp_pack_twice_four:wNNNNNNNN
28090   \__fp_use_i:ww , \__fp_sep:
28091   #1 #2 0000 0000 0000 0000 0000 0000 \__fp_sep:
28092 }
28093 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1\__fp_sep: #2\__fp_sep: #3\__fp_sep:
28094 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} \__fp_sep: }

```

(End of definition for `\__fp_ep_to_ep:wwN` and others.)

`\__fp_ep_compare:www`
`\__fp_ep_compare_aux:www`

In `l3fp-trig` we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only

16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000, 9999].

```

28095 \cs_new:Npn \__fp_ep_compare:www #1,#2#3#4#5#6#7\__fp_sep:
28096 { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}\__fp_sep: #6#7\__fp_sep: }
28097 \cs_new:Npn \__fp_ep_compare_aux:www
28098 #1\__fp_sep:#2\__fp_sep:#3,#4#5#6#7#8#9\__fp_sep:
28099 {
28100   \if_case:w
28101     \__fp_compare_npos:nwnw
28102     #1\__fp_sep: {#3}{#4}{#5}{#6}{#7}\__fp_sep: \exp_stop_f:
28103     \if_int_compare:w #2 = #8#9 \exp_stop_f:
28104       0
28105     \else:
28106       \if_int_compare:w #2 < #8#9 - \fi: 1
28107     \fi:
28108   \or: 1
28109   \else: -1
28110   \fi:
28111 }

```

(End of definition for __fp_ep_compare:www and __fp_ep_compare_aux:www.)

__fp_ep_mul:wwwN
__fp_ep_mul_raw:wwwN

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100, 9999].

```

28112 \cs_new:Npn \__fp_ep_mul:wwwN #1,#2\__fp_sep: #3,#4\__fp_sep:
28113 {
28114   \__fp_ep_to_ep:wwN #3,#4\__fp_sep:
28115   \__fp_fixed_continue:wn
28116   {
28117     \__fp_ep_to_ep:wwN #1,#2\__fp_sep:
28118     \__fp_ep_mul_raw:wwwN
28119   }
28120   \__fp_fixed_continue:wn
28121 }
28122 \cs_new:Npn \__fp_ep_mul_raw:wwwN #1,#2\__fp_sep: #3,#4\__fp_sep: #5
28123 {
28124   \__fp_fixed_mul:wn #2\__fp_sep: #4\__fp_sep:
28125   { \exp_after:wN #5 \int_value:w \__fp_int_eval:w #1 + #3 , }
28126 }

```

(End of definition for __fp_ep_mul:wwwN and __fp_ep_mul_raw:wwwN.)

80.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8/\langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left\lfloor \frac{10^9}{\langle d_1 \rangle + 1} \right\rfloor \\ \beta &= \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor \\ a &= 10^3\alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,\end{aligned}$$

where $\left\lfloor \frac{\cdot}{\cdot} \right\rfloor$ denotes ε -TeX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2 \rangle/10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1 \rangle \simeq 10^8/\langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1 \rangle + 1) \simeq 10^8/\langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a/10^8 = 1 - \epsilon$ using the relation $1/(1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d \rangle < 10^3\langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TeX's division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ underestimates $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7\langle d \rangle a < \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3\langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle/10] + a)(b - c[\langle d_2 \rangle/10]) \leq (b + ca)^2/(4c)$. Hence,

$$10^7\langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8} \cdot 10^{-9}\langle d_1 \rangle(\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle(\langle d_1 \rangle + 1)$, hence $10^7\langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\dots$, and going back through the derivation of

the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`\__fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle \text{continuation} \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `\__fp_ep_div_esti:wwwn` $\langle \text{denominator} \rangle$ $\langle \text{numerator} \rangle$, responsible for estimating the inverse of the denominator.

```

28127 \cs_new:Npn \__fp_ep_div:wwwn #1,#2\__fp_sep: #3,#4\__fp_sep:
28128 {
28129   \__fp_ep_to_ep:wwN #1,#2\__fp_sep:
28130   \__fp_fixed_continue:wn
28131   {
28132     \__fp_ep_to_ep:wwN #3,#4\__fp_sep:
28133     \__fp_ep_div_esti:wwwn
28134   }
28135 }
```

(End of definition for `\__fp_ep_div:wwwn`.)

`\__fp_ep_div_esti:wwwn` The `esti` function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute $\langle n \rangle / (10 \langle d \rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result $10^{-8}a \langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `\__fp_ep_div_epsilon:wnNNNNn`, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator `#8`.

```

28136 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3\__fp_sep: #4,
28137 {
28138   \exp_after:wN \__fp_ep_div_estii:wwnnwwn
28139   \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
28140   \exp_after:wN \__fp_sep:
28141   \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
28142   {#2} #3\__fp_sep:
28143 }
28144 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn
28145 #1\__fp_sep: #2,#3#4#5\__fp_sep: #6\__fp_sep: #7
```

```

28146 {
28147   \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
28148   \int_value:w \__fp_int_eval:w 10 0000 0000 - 1750
28149   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) \__fp_sep:
28150   {#3}{#4}#5\__fp_sep: #6\__fp_sep: { #7 #2, }
28151 }
28152 \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6\__fp_sep: #7\__fp_sep:
28153 {
28154   \__fp_fixed_mul_short:wwn #7\__fp_sep: {#1}{#2#3#4#5}{#6}\__fp_sep:
28155   \__fp_ep_div_epsilon:wnNNNNNn {#1#2#3#4}#5#6
28156   \__fp_fixed_mul:wwn
28157 }

```

(End of definition for __fp_ep_div_esti:wwwwn, __fp_ep_div_estii:wwnnwwn, and __fp_ep_div_estiii:NNNNNwwwn.)

__fp_ep_div_epsilon:wnNNNNNn
 __fp_ep_div_eps_pack:NNNNNw
 __fp_ep_div_epsii:wwnnNNNNNn

The bounds shown above imply that the `epsi` function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsii` evaluates $10^{-9}a/(1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

28158 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6\__fp_sep:
28159 {
28160   \exp_after:wN \__fp_ep_div_epsii:wwnnNNNNNn
28161   \int_value:w \__fp_int_eval:w 1 9998 - #2
28162   \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
28163   \int_value:w \__fp_int_eval:w 1 9999 9998 - #3#4
28164   \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
28165   \int_value:w \__fp_int_eval:w 2 0000 0000 - #5#6 \__fp_sep: \__fp_sep:
28166 }
28167 \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6\__fp_sep:
28168 { + #1 \__fp_sep: {#2#3#4#5} {#6} }
28169 \cs_new:Npn \__fp_ep_div_epsii:wwnnNNNNNn 1#1\__fp_sep: #2\__fp_sep: #3#4#5#6#7#8
28170 {
28171   \__fp_fixed_mul:wwn {0000}{#1}#2\__fp_sep: {0000}{#1}#2\__fp_sep:
28172   \__fp_fixed_add_one:wN
28173   \__fp_fixed_mul:wwn {10000} {#1} #2 \__fp_sep:
28174   {
28175     \__fp_fixed_mul_short:wwn
28176     {0000}{#1}#2\__fp_sep: {#3}{#4#5#6#7}{#8000}\__fp_sep:
28177     \__fp_fixed_div_myriad:wn
28178     \__fp_fixed_mul:wwn
28179   }
28180   \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000}\__fp_sep:
28181 }

```

(End of definition for __fp_ep_div_epsilon:wnNNNNNn, __fp_ep_div_eps_pack:NNNNNw, and __fp_ep_div_epsii:wwnnNNNNNn.)

80.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

```

    \_fp_ep_isqrt:wwn
    \_fp_ep_isqrt_aux:wwn
    \_fp_ep_isqrt_auxii:wwnnwn

```

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be $-\#1/2$, otherwise it will be $(\#1 - 1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa ($\#5 \in [1000, 9999]$), and as #7 the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses #2 and 0, an approximation #5 of $10^4 x$ (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

28182 \cs_new:Npn \_fp_ep_isqrt:wwn #1,#2\_fp_sep:
28183 {
28184   \_fp_ep_to_ep:wwN #1,#2\_fp_sep:
28185   \_fp_ep_isqrt_auxi:wwn
28186 }
28187 \cs_new:Npn \_fp_ep_isqrt_auxi:wwn #1,
28188 {
28189   \exp_after:wN \_fp_ep_isqrt_auxii:wwnnwn
28190   \int_value:w \_fp_int_eval:w
28191   \int_if_odd:nTF {#1}
28192     { (1 - #1) / 2 , 535 , { 0 } { } }
28193     { 1 - #1 / 2 , 168 , { } { 0 } }
28194 }
28195 \cs_new:Npn \_fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6\_fp_sep: #7
28196 {
28197   \_fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
28198   {#5} #6 \_fp_sep: { #7 #1 , }
28199 }

```

(End of definition for `\_fp_ep_isqrt:wwn`, `\_fp_ep_isqrt_aux:wwn`, and `\_fp_ep_isqrt_auxii:wwnnwn`.)

```

    \_fp_ep_isqrt_esti:wwnnwn
    \_fp_ep_isqrt_estii:wwnnwn
    \_fp_ep_isqrt_estiii:NNNNwwn

```

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle \text{prev} \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle \text{prev} \rangle \cdot x))/2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check

by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to **esti**. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to **__fp_ep_isqrt_epsilon:wN**, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

28200 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
28201 {
28202   \if_int_compare:w #1 = #2 \exp_stop_f:
28203   \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
28204   \fi:
28205   \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
28206   \int_value:w \__fp_int_eval:w
28207   (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
28208   #1, #3, {#4}
28209 }
28210 \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
28211 {
28212   \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwwn
28213   \int_value:w \__fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
28214   \exp_after:wN , \int_value:w \__fp_int_eval:w 10000 + #2 \__fp_sep:
28215 }
28216 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn
28217 1#1#2#3#4#5#6, 1#7#8\__fp_sep: #9\__fp_sep:
28218 {
28219   \__fp_fixed_mul_short:wwn #9\__fp_sep: {#1} {#2#3#4#5} {#600} \__fp_sep:
28220   \__fp_ep_isqrt_epsilon:wN
28221   \__fp_fixed_mul_short:wwn {#7} {#80} {0000} \__fp_sep:
28222 }

```

(End of definition for **__fp_ep_isqrt_esti:wwnnwn**, **__fp_ep_isqrt_estii:wwnnwn**, and **__fp_ep_isqrt_estiii:NNNNNwwwn**.)

__fp_ep_isqrt_epsilon:wN Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The **epsii** auxiliary receives z as #1 and y as #2.

```

28223 \cs_new:Npn \__fp_ep_isqrt_epsilon:wN #1\__fp_sep:
28224 {
28225   \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}\__fp_sep: #1\__fp_sep:
28226   \__fp_ep_isqrt_epsii:wwN #1\__fp_sep:
28227   \__fp_ep_isqrt_epsii:wwN #1\__fp_sep:
28228   \__fp_ep_isqrt_epsii:wwN #1\__fp_sep:
28229 }
28230 \cs_new:Npn \__fp_ep_isqrt_epsii:wwN #1\__fp_sep: #2\__fp_sep:
28231 {
28232   \__fp_fixed_mul:wwn #1\__fp_sep: #1\__fp_sep:
28233   \__fp_fixed_mul_sub_back:wwn #2\__fp_sep:
28234   {15000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
28235   \__fp_fixed_mul:wwn #1\__fp_sep:
28236 }

```

(End of definition for **__fp_ep_isqrt_epsilon:wN** and **__fp_ep_isqrt_epsii:wwN**.)

80.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`\__fp_ep_to_float_o:wwN`
`\__fp_ep_inv_to_float_o:wwN` An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
28237 \cs_new:Npn \__fp_ep_to_float_o:wwN #1,
28238 { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wN }
28239 \cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2\__fp_sep:
28240 {
28241   \__fp_ep_div:wwwn
28242   1,{1000}{0000}{0000}{0000}{0000}{0000}\__fp_sep: #1,#2\__fp_sep:
28243   \__fp_ep_to_float_o:wwN
28244 }
```

(End of definition for `\__fp_ep_to_float_o:wwN` and `\__fp_ep_inv_to_float_o:wwN`.)

`\__fp_fixed_inv_to_float_o:wN` Another function which reduces to converting an extended precision number to a float.

```
28245 \cs_new:Npn \__fp_fixed_inv_to_float_o:wN
28246 { \__fp_ep_inv_to_float_o:wwN 0, }
```

(End of definition for `\__fp_fixed_inv_to_float_o:wN`.)

`\__fp_fixed_to_float_rad_o:wN` Converts the fixed point number `#1` from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```
28247 \cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1\__fp_sep:
28248 {
28249   \__fp_fixed_mul:wwn #1\__fp_sep: {5729}{5779}{5130}{8232}{0876}{7981}\__fp_sep:
28250   { \__fp_ep_to_float_o:wwN 2, }
28251 }
```

(End of definition for `\__fp_fixed_to_float_rad_o:wN`.)

`\__fp_fixed_to_float_o:wN`
`\__fp_fixed_to_float_o:Nw` ... `\__fp_int_eval:w` $\langle exponent \rangle$ `\__fp_fixed_to_float_o:wN` $\{\langle a_1 \rangle\}$ $\{\langle a_2 \rangle\}$ $\{\langle a_3 \rangle\}$ $\{\langle a_4 \rangle\}$ $\{\langle a_5 \rangle\}$ $\{\langle a_6 \rangle\}$ `\__fp_sep:` $\langle sign \rangle$
yields

$\langle exponent' \rangle$ `\__fp_sep:` $\{\langle a'_1 \rangle\}$ $\{\langle a'_2 \rangle\}$ $\{\langle a'_3 \rangle\}$ $\{\langle a'_4 \rangle\}$ `\__fp_sep:`

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .¹²

```
28252 \cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2\__fp_sep:
28253 { \__fp_fixed_to_float_o:wN #2\__fp_sep: #1 }
28254 \cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6\__fp_sep: #7
28255 { % for the 8-digit-at-the-start thing
28256   + \__fp_int_eval:w \c__fp_block_int
28257   \exp_after:wN \exp_after:wN
```

¹²Bruno: I must double check this assumption.

```

28258 \exp_after:wN \__fp_fixed_to_loop:N
28259 \exp_after:wN \use_none:n
28260 \int_value:w \__fp_int_eval:w
28261 1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
28262 \int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
28263 \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
28264 \int_value:w 1#5#6
28265 \exp_after:wN \__fp_sep:
28266 \exp_after:wN \__fp_sep:
28267 }
28268 \cs_new:Npn \__fp_fixed_to_loop:N #1
28269 {
28270 \if_meaning:w 0 #1
28271 - 1
28272 \exp_after:wN \__fp_fixed_to_loop:N
28273 \else:
28274 \exp_after:wN \__fp_fixed_to_loop_end:w
28275 \exp_after:wN #1
28276 \fi:
28277 }
28278 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 \__fp_sep:
28279 {
28280 \if_meaning:w \__fp_sep: #1
28281 \exp_after:wN \__fp_fixed_to_float_zero:w
28282 \else:
28283 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
28284 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
28285 \exp_after:wN \__fp_fixed_to_float_pack:ww
28286 \exp_after:wN \__fp_sep:
28287 \fi:
28288 #1 #2 0000 0000 0000 0000 \__fp_sep:
28289 }
28290 \cs_new:Npn \__fp_fixed_to_float_zero:w \__fp_sep: 0000 0000 0000 0000 \__fp_sep:
28291 {
28292 - 2 * \c__fp_max_exponent_int \__fp_sep:
28293 {0000} {0000} {0000} {0000} \__fp_sep:
28294 }
28295 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 \__fp_sep: #2#3 \__fp_sep: \__fp_sep:
28296 {
28297 \if_int_compare:w #2 > 4 \exp_stop_f:
28298 \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
28299 \fi:
28300 \__fp_sep: #1 \__fp_sep:
28301 }
28302 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw \__fp_sep: #1#2#3#4 \__fp_sep:
28303 {
28304 \exp_after:wN \__fp_basics_pack_high:NNNNw
28305 \int_value:w \__fp_int_eval:w 1 #1#2
28306 \exp_after:wN \__fp_basics_pack_low:NNNNw
28307 \int_value:w \__fp_int_eval:w 1 #3#4 + 1 \__fp_sep:
28308 }

```

(End of definition for __fp_fixed_to_float_o:wN and __fp_fixed_to_float_o:Nw.)

28309 `</code>`

Chapter 81

l3fp-expo implementation

```
28310 ⟨*code⟩
28311 ⟨@@=fp⟩

\__fp_parse_word_exp:N Unary functions.
\__fp_parse_word_ln:N 28312 \cs_new:Npn \__fp_parse_word_exp:N
\__fp_parse_word_fact:N 28313 { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
28314 \cs_new:Npn \__fp_parse_word_ln:N
28315 { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
28316 \cs_new:Npn \__fp_parse_word_fact:N
28317 { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }

(End of definition for \__fp_parse_word_exp:N, \__fp_parse_word_ln:N, and \__fp_parse_word_fact:N.)
```

81.1 Logarithm

81.1.1 Work plan

As for many other functions, we filter out special cases in `\__fp_ln_o:w`. Then `\__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of $\ln$ near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

81.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```

\c__fp_ln_i_fixed_tl 28318 \tl_const:Nn \c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl 28319 { {0000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:}
\c__fp_ln_iii_fixed_tl 28320 \tl_const:Nn \c__fp_ln_ii_fixed_tl
\c__fp_ln_iv_fixed_tl 28321 { {6931}{4718}{0559}{9453}{0941}{7232}\__fp_sep:}
\c__fp_ln_vii_fixed_tl 28322 \tl_const:Nn \c__fp_ln_iii_fixed_tl
\c__fp_ln_viii_fixed_tl 28323 { {10986}{1228}{8668}{1096}{9139}{5245}\__fp_sep:}
\c__fp_ln_ix_fixed_tl 28324 \tl_const:Nn \c__fp_ln_iv_fixed_tl
\c__fp_ln_x_fixed_tl 28325 { {13862}{9436}{1119}{8906}{1883}{4464}\__fp_sep:}
28326 \tl_const:Nn \c__fp_ln_vi_fixed_tl
28327 { {17917}{5946}{9228}{0550}{0081}{2477}\__fp_sep:}
28328 \tl_const:Nn \c__fp_ln_vii_fixed_tl
28329 { {19459}{1014}{9055}{3133}{0510}{5353}\__fp_sep:}
28330 \tl_const:Nn \c__fp_ln_viii_fixed_tl
28331 { {20794}{4154}{1679}{8359}{2825}{1696}\__fp_sep:}
28332 \tl_const:Nn \c__fp_ln_ix_fixed_tl
28333 { {21972}{2457}{7336}{2193}{8279}{0490}\__fp_sep:}
28334 \tl_const:Nn \c__fp_ln_x_fixed_tl
28335 { {23025}{8509}{2994}{0456}{8401}{7991}\__fp_sep:}

```

(End of definition for `\c__fp_ln_i_fixed_tl` and others.)

81.1.3 Sign, exponent, and special numbers

`\__fp_ln_o:w` The logarithm of negative numbers (including $-\infty$ and -0) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `\__fp_ln_npos_o:w`.

```

28336 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
28337 {
28338   \if_meaning:w 2 #3
28339     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
28340   \fi:
28341   \if_case:w #2 \exp_stop_f:
28342     \__fp_case_use:nw
28343     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
28344   \or:
28345   \else:
28346     \__fp_case_return_same_o:w
28347   \fi:
28348   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4\__fp_sep:
28349 }

```

(End of definition for `\__fp_ln_o:w`.)

81.1.4 Absolute ln

`\__fp_ln_npos_o:w` We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4} , and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```

28350 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3\__fp_sep:
28351 { %^A todo: ln(1) should be "exact zero", not "underflow"
28352   \exp_after:wN \__fp_sanitize:Nw
28353   \int_value:w % for the overall sign
28354   \if_int_compare:w #1 < \c_one_int
28355     2
28356   \else:
28357     0
28358   \fi:
28359   \exp_after:wN \exp_stop_f:
28360   \int_value:w \__fp_int_eval:w % for the exponent
28361   \__fp_ln_significand:NNNNnnnnN #2#3
28362   \__fp_ln_exponent:wn {#1}
28363 }

```

(End of definition for `\__fp_ln_npos_o:w`.)

`\__fp_ln_significand:NNNNnnnnN` `\__fp_ln_significand:NNNNnnnnN` $\langle X_1 \rangle$ $\{\langle X_2 \rangle\}$ $\{\langle X_3 \rangle\}$ $\{\langle X_4 \rangle\}$ $\langle continuation \rangle$
This function expands to

$\langle continuation \rangle$ $\{\langle Y_1 \rangle\}$ $\{\langle Y_2 \rangle\}$ $\{\langle Y_3 \rangle\}$ $\{\langle Y_4 \rangle\}$ $\{\langle Y_5 \rangle\}$ $\{\langle Y_6 \rangle\}$ `\__fp_sep:`

where $Y = -\ln(X)$ as an extended fixed point.

```

28364 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
28365 {
28366   \exp_after:wN \__fp_ln_x_ii:wnnnn
28367   \int_value:w
28368   \if_case:w #1 \exp_stop_f:
28369   \or:
28370     \if_int_compare:w #2 < 4 \exp_stop_f:
28371     \__fp_int_eval:w 10 - #2
28372   \else:
28373     6
28374   \fi:
28375   \or: 4
28376   \or: 3
28377   \or: 2
28378   \or: 2
28379   \or: 2
28380   \else: 1
28381   \fi:
28382   \__fp_sep: { #1 #2 #3 #4 }
28383 }

```

(End of definition for `\__fp_ln_significand:NNNNnnnnN`.)

`\__fp_ln_x_ii:wnnnn` We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

28384 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1\__fp_sep: #2#3#4#5
28385 {
28386   \exp_after:wN \__fp_ln_div_after:Nw

```

```

28387 \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
28388 \int_value:w
28389 \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
28390 \int_value:w \__fp_int_eval:w
28391 \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
28392 \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
28393 \exp_after:wN \__fp_ln_x_iii:NNNNNw
28394 \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 \__fp_sep:
28395 {20000} {0000} {0000} {0000}
28396 } %^A todo: reoptimize (a generalization attempt failed).
28397 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7\__fp_sep:
28398 { #1#2\__fp_sep: {#3#4#5#6} {#7} }
28399 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6\__fp_sep:
28400 {
28401 #1#2#3#4#5 + 1 \__fp_sep:
28402 {#1#2#3#4#5} {#6}
28403 }

```

The Taylor series to be used is expressed in terms of $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$. We now compute the quotient with extended precision, reusing some code from `\__fp_-/_o:ww`. Note that $1 + x$ is known exactly.

To reuse notations from `l3fp-basics`, we want to compute A/Z with $A = 2$ and $Z = x + 1$. In `l3fp-basics`, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A , B , C , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147 \dots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4 , and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The $1/2$ comes from how ε -TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, i.e., Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned}
10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
&\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
&\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
\end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
\end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> __fp_sep: {<4d>} {<4d>} <fixed-t1>`

The number is x . Compute y by adding 1 to the five first digits.

```

28404 \cs_new:Npn __fp_ln_x_iv:wnnnnnnnn #1__fp_sep: #2#3#4#5 #6#7#8#9
28405 {
28406   \exp_after:wN __fp_div_significand_pack:NNN
28407   \int_value:w __fp_int_eval:w
28408   __fp_ln_div_i:w #1 __fp_sep:
28409     #6 #7 __fp_sep: {#8} {#9}
28410     {#2} {#3} {#4} {#5}
28411     { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
28412     { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
28413     { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
28414     { \exp_after:wN __fp_ln_div_ii:wnn \int_value:w #1 }
28415     { \exp_after:wN __fp_ln_div_vi:wnn \int_value:w #1 }
28416 }
28417 \cs_new:Npn __fp_ln_div_i:w #1__fp_sep:
28418 {
28419   \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
28420   \int_value:w __fp_int_eval:w 999999 + 2 0000 0000 / #1 __fp_sep: % Q1
28421 }
28422 \cs_new:Npn __fp_ln_div_ii:wnn
28423   #1__fp_sep: #2__fp_sep:#3 % y__fp_sep: B1__fp_sep:B2 <- for k=1
28424 {
28425   \exp_after:wN __fp_div_significand_pack:NNN
28426   \int_value:w __fp_int_eval:w
28427     \exp_after:wN __fp_div_significand_calc:wnnnnnnnn
28428     \int_value:w __fp_int_eval:w 999999 + #2 #3 / #1 __fp_sep: % Q2
28429     #2 #3 __fp_sep:
28430 }
28431 \cs_new:Npn __fp_ln_div_vi:wnn
28432   #1__fp_sep: #2__fp_sep:#3#4#5 #6#7#8#9 %y__fp_sep:F1__fp_sep:F2F3F4x1x2x3x4

```

```

28433 {
28434   \exp_after:wN \__fp_div_significand_pack:NNN
28435   \int_value:w \__fp_int_eval:w 1000000 + #2 #3 / #1 \__fp_sep: % Q6
28436 }

```

We now have essentially

```

\__fp_ln_div_after:Nw <fixed t1>
\__fp_div_significand_pack:NNN 106 + Q1
\__fp_div_significand_pack:NNN 106 + Q2
\__fp_div_significand_pack:NNN 106 + Q3
\__fp_div_significand_pack:NNN 106 + Q4
\__fp_div_significand_pack:NNN 106 + Q5
\__fp_div_significand_pack:NNN 106 + Q6 \__fp_sep:
<exponent> \__fp_sep: <continuation>

```

where $\langle \text{fixed } t1 \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then `\__fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put `\__fp_sep:s` between each piece. Once those have been expanded, we get

```

\__fp_ln_div_after:Nw <fixed-t1> <1d> \__fp_sep: <4d> \__fp_sep:
<4d> \__fp_sep: <4d> \__fp_sep: <4d> \__fp_sep: <4d> \__fp_sep:
<4d> \__fp_sep: <exponent> \__fp_sep:

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

28437 \cs_new:Npn \__fp_ln_div_after:Nw #1#2\__fp_sep:
28438 {
28439   \if_meaning:w 0 #2
28440     \exp_after:wN \__fp_ln_t_small:Nw
28441   \else:
28442     \exp_after:wN \__fp_ln_t_large:NNw
28443     \exp_after:wN -
28444   \fi:
28445   #1
28446 }
28447 \cs_new:Npn \__fp_ln_t_small:Nw
28448   #1 #2\__fp_sep: #3\__fp_sep: #4\__fp_sep: #5\__fp_sep: #6\__fp_sep: #7\__fp_sep:
28449 {
28450   \exp_after:wN \__fp_ln_t_large:NNw
28451   \exp_after:wN + % <sign>
28452   \exp_after:wN #1
28453   \int_value:w \__fp_int_eval:w 9999 - #2 \exp_after:wN \__fp_sep:
28454   \int_value:w \__fp_int_eval:w 9999 - #3 \exp_after:wN \__fp_sep:
28455   \int_value:w \__fp_int_eval:w 9999 - #4 \exp_after:wN \__fp_sep:
28456   \int_value:w \__fp_int_eval:w 9999 - #5 \exp_after:wN \__fp_sep:
28457   \int_value:w \__fp_int_eval:w 9999 - #6 \exp_after:wN \__fp_sep:
28458   \int_value:w \__fp_int_eval:w 1 0000 - #7 \__fp_sep:
28459 }

\__fp_ln_t_large:NNw <sign> <fixed t1>
<t1>\__fp_sep: <t2> \__fp_sep: <t3>\__fp_sep: <t4>
\__fp_sep: <t5> \__fp_sep: <t6>\__fp_sep:
<exponent> \__fp_sep: <continuation>

```

Compute the square t^2 , and keep t at the end with its sign. We know that $t < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `\__fp_ln_t_small:w`, they can have less than 4 digits.

```

28460 \cs_new:Npn \__fp_ln_t_large:NNw
28461   #1 #2
28462   #3\__fp_sep: #4\__fp_sep: #5\__fp_sep: #6\__fp_sep: #7\__fp_sep: #8\__fp_sep:
28463   {
28464     \exp_after:wN \__fp_ln_square_t_after:w
28465     \int_value:w \__fp_int_eval:w 9999 0000 + #3*#3
28466     \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
28467     \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#4
28468     \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
28469     \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
28470     \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
28471     \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
28472     \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
28473     \int_value:w \__fp_int_eval:w
28474       1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
28475       + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
28476       % \__fp_sep: \__fp_sep: \__fp_sep:
28477     \exp_after:wN \__fp_ln_twice_t_after:w
28478     \int_value:w \__fp_int_eval:w -1 + 2*#3
28479     \exp_after:wN \__fp_ln_twice_t_pack:Nw
28480     \int_value:w \__fp_int_eval:w 9999 + 2*#4
28481     \exp_after:wN \__fp_ln_twice_t_pack:Nw
28482     \int_value:w \__fp_int_eval:w 9999 + 2*#5
28483     \exp_after:wN \__fp_ln_twice_t_pack:Nw
28484     \int_value:w \__fp_int_eval:w 9999 + 2*#6
28485     \exp_after:wN \__fp_ln_twice_t_pack:Nw
28486     \int_value:w \__fp_int_eval:w 9999 + 2*#7
28487     \exp_after:wN \__fp_ln_twice_t_pack:Nw
28488     \int_value:w \__fp_int_eval:w 10000 + 2*#8 \__fp_sep: \__fp_sep:
28489     { \__fp_ln_c:NwNw #1 }
28490     #2
28491   }
28492 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2\__fp_sep: { + #1 \__fp_sep: {#2} }
28493 \cs_new:Npn \__fp_ln_twice_t_after:w #1\__fp_sep:
28494   { \__fp_sep:\__fp_sep:\__fp_sep: {#1} }
28495 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6\__fp_sep:
28496   { + #1#2#3#4#5 \__fp_sep: {#6} }
28497 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4\__fp_sep:
28498   { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End of definition for `\__fp_ln_x_ii:wnnnn`.)

`\__fp_ln_Taylor:wwNw` Denoting $T = t^2$, we get

```

\__fp_ln_Taylor:wwNw
{<T1>} {<T2>} {<T3>} {<T4>} {<T5>} {<T6>} \__fp_sep: \__fp_sep:
{<(2t)1>} {<(2t)2>} {<(2t)3>} {<(2t)4>} {<(2t)5>} {<(2t)6>} \__fp_sep:
{ \__fp_ln_c:NwNw <sign> }
<fixed t1> <exponent> \__fp_sep: <continuation>

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T\left(\frac{1}{3} + T\left(\frac{1}{5} + T\left(\frac{1}{7} + T\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows (_fp_sep: represented by ;)

```
\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;
```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
28499 \cs_new:Npn \__fp_ln_Taylor:wwNw
28500 {
28501   \__fp_ln_Taylor_loop:www
28502   21 \__fp_sep: {0000}{0000}{0000}{0000}{0000}{0000} \__fp_sep:
28503 }
28504 \cs_new:Npn \__fp_ln_Taylor_loop:www #1\__fp_sep: #2\__fp_sep: #3\__fp_sep:
28505 {
28506   \if_int_compare:w #1 = \c_one_int
28507   \__fp_ln_Taylor_break:w
28508   \fi:
28509   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1\__fp_sep:
28510   \__fp_fixed_add:wwN #2\__fp_sep:
28511   \__fp_fixed_mul:wwN #3\__fp_sep:
28512   {
28513     \exp_after:wN \__fp_ln_Taylor_loop:www
28514     \int_value:w \__fp_int_eval:w #1 - 2 \__fp_sep:
28515   }
28516   #3\__fp_sep:
28517 }
28518 \cs_new:Npn \__fp_ln_Taylor_break:w
28519 \fi: #1 \__fp_fixed_add:wwN #2#3\__fp_sep: #4 \__fp_sep:\__fp_sep:
28520 {
28521   \fi:
28522   \exp_after:wN \__fp_fixed_mul:wwN
28523   \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3\__fp_sep:
28524 }
```

(End of definition for __fp_ln_Taylor:wwNw.)

```
\__fp_ln_c:NwNw \sign
{\langle r_1 \rangle \langle r_2 \rangle \langle r_3 \rangle \langle r_4 \rangle \langle r_5 \rangle \langle r_6 \rangle \__fp_sep:
\fixed_tl \exponent \__fp_sep: \continuation}
```

We are now reduced to finding $\ln(c)$ and $\langle \text{exponent} \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $b \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```
28525 \cs_new:Npn \__fp_ln_c:NwNw #1 #2\__fp_sep: #3
28526 {
28527   \if_meaning:w + #1
```



```

28528     \exp_after:wN \exp_after:wN \exp_after:wN \_\_fp\_fixed\_sub:wwn
28529   \else:
28530     \exp_after:wN \exp_after:wN \exp_after:wN \_\_fp\_fixed\_add:wwn
28531   \fi:
28532   #3 #2 \_\_fp\_sep:
28533 }

```

(End of definition for `\_\_fp\_ln\_c:NwNw`.)

```

\_\_fp\_ln\_exponent:wn
  \_\_fp\_ln\_exponent:wn
  {\langle s_1 \rangle} {\langle s_2 \rangle} {\langle s_3 \rangle} {\langle s_4 \rangle} {\langle s_5 \rangle} {\langle s_6 \rangle} \_\_fp\_sep:
  {\langle exponent \rangle}

```

Compute $\langle \text{exponent} \rangle$ times $\ln(10)$. Apart from the cases where $\langle \text{exponent} \rangle$ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

28534 \cs_new:Npn \_\_fp\_ln\_exponent:wn #1\_\_fp\_sep: #2
28535 {
28536   \if_case:w #2 \exp_stop_f:
28537     0 \_\_fp\_case\_return:nw { \_\_fp\_fixed\_to\_float\_o:Nw 2 }
28538   \or:
28539     \exp_after:wN \_\_fp\_ln\_exponent\_one:ww \int\_value:w
28540   \else:
28541     \if_int_compare:w #2 > \c\_zero\_int
28542       \exp_after:wN \_\_fp\_ln\_exponent\_small:NNww
28543       \exp_after:wN 0
28544       \exp_after:wN \_\_fp\_fixed\_sub:wwn \int\_value:w
28545     \else:
28546       \exp_after:wN \_\_fp\_ln\_exponent\_small:NNww
28547       \exp_after:wN 2
28548       \exp_after:wN \_\_fp\_fixed\_add:wwn \int\_value:w -
28549     \fi:
28550   \fi:
28551   #2\_\_fp\_sep: #1\_\_fp\_sep:
28552 }

```

Now we painfully write all the cases.¹³ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

28553 \cs_new:Npn \_\_fp\_ln\_exponent\_one:ww 1\_\_fp\_sep: #1\_\_fp\_sep:
28554 {
28555   0
28556   \exp_after:wN \_\_fp\_fixed\_sub:wwn \c\_fp\_ln\_x\_fixed\_t1 #1\_\_fp\_sep:
28557   \_\_fp\_fixed\_to\_float\_o:wN 0
28558 }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

28559 \cs_new:Npn \_\_fp\_ln\_exponent\_small:NNww #1#2#3\_\_fp\_sep: #4#5#6#7#8#9\_\_fp\_sep:

```

¹³Bruno: do rounding.

```

28560 {
28561     4
28562     \exp_after:wN \__fp_fixed_mul:wnn
28563     \c__fp_ln_x_fixed_t1
28564     {#3}{0000}{0000}{0000}{0000}{0000} \__fp_sep:
28565     #2
28566     {0000}{#4}{#5}{#6}{#7}{#8}\__fp_sep:
28567     \__fp_fixed_to_float_o:wN #1
28568 }

```

(End of definition for __fp_ln_exponent:wn.)

81.2 Exponential

81.2.1 Sign, exponent, and special numbers

__fp_exp_o:w

```

28569 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
28570 {
28571     \if_case:w #2 \exp_stop_f:
28572     \__fp_case_return_o:Nw \c_one_fp
28573     \or:
28574     \exp_after:wN \__fp_exp_normal_o:w
28575     \or:
28576     \if_meaning:w 0 #3
28577     \exp_after:wN \__fp_case_return_o:Nw
28578     \exp_after:wN \c_inf_fp
28579     \else:
28580     \exp_after:wN \__fp_case_return_o:Nw
28581     \exp_after:wN \c_zero_fp
28582     \fi:
28583     \or:
28584     \__fp_case_return_same_o:w
28585     \fi:
28586     \s__fp \__fp_chk:w #2#3#4\__fp_sep:
28587 }

```

(End of definition for __fp_exp_o:w.)

__fp_exp_normal_o:w

__fp_exp_pos_o:NNwnw

__fp_exp_overflow:NN

```

28588 \cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
28589 {
28590     \if_meaning:w 0 #1
28591     \__fp_exp_pos_o:NNwnw + \__fp_fixed_to_float_o:wN
28592     \else:
28593     \__fp_exp_pos_o:NNwnw - \__fp_fixed_inv_to_float_o:wN
28594     \fi:
28595 }
28596 \cs_new:Npn \__fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5\__fp_sep:
28597 {
28598     \fi:
28599     \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
28600     \token_if_eq_charcode:NNTF + #1
28601     { \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp }

```

```

28602     { \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp }
28603     \exp:w
28604   \else:
28605     \exp_after:wN \__fp_sanitize:Nw
28606     \exp_after:wN 0
28607     \int_value:w #1 \__fp_int_eval:w
28608     \if_int_compare:w #4 < \c_zero_int
28609       \exp_after:wN \use_i:nn
28610     \else:
28611       \exp_after:wN \use_ii:nn
28612     \fi:
28613     {
28614       0
28615       \__fp_decimate:nNnnnn { - #4 }
28616       \__fp_exp_Taylor:Nnnwn
28617     }
28618     {
28619       \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
28620       \__fp_exp_pos_large:NnnNwn
28621     }
28622     #5
28623     {#4}
28624     #1 #2 0
28625     \exp:w
28626   \fi:
28627   \exp_after:wN \exp_end:
28628 }
28629 \cs_new:Npn \__fp_exp_overflow:NN #1#2
28630 {
28631   \exp_after:wN \exp_after:wN
28632   \exp_after:wN #1
28633   \exp_after:wN #2
28634 }

```

(End of definition for __fp_exp_normal_o:w, __fp_exp_pos_o:Nnnwn, and __fp_exp_overflow:NN.)

```

\__fp_exp_Taylor:Nnnwn
\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a __fp_sep:, form a fixed point number, so we pack it in blocks of 4 digits.

```

28635 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4\__fp_sep: #5 #6
28636 {
28637   #6
28638   \__fp_pack_twice_four:wNNNNNNNN
28639   \__fp_pack_twice_four:wNNNNNNNN
28640   \__fp_pack_twice_four:wNNNNNNNN
28641   \__fp_exp_Taylor_ii:ww
28642   \__fp_sep: #2#3#4 0000 0000 \__fp_sep:
28643 }
28644 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1\__fp_sep: #2\__fp_sep:
28645 { \__fp_exp_Taylor_loop:www 10 \__fp_sep: #1 \__fp_sep: #1 \__fp_sep: \s_fp_stop }
28646 \cs_new:Npn \__fp_exp_Taylor_loop:www #1\__fp_sep: #2\__fp_sep: #3\__fp_sep:
28647 {
28648   \if_int_compare:w #1 = \c_one_int

```

```

28649     \exp_after:wN \__fp_exp_Taylor_break:Nww
28650 \fi:
28651 \__fp_fixed_div_int:wwN #3 \__fp_sep: #1 \__fp_sep:
28652 \__fp_fixed_add_one:wN
28653 \__fp_fixed_mul:wwN #2 \__fp_sep:
28654 {
28655     \exp_after:wN \__fp_exp_Taylor_loop:www
28656     \int_value:w \__fp_int_eval:w #1 - 1 \__fp_sep:
28657     #2 \__fp_sep:
28658 }
28659 }
28660 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2\__fp_sep: #3 \s__fp_stop
28661 { \__fp_fixed_add_one:wN #2 \__fp_sep: }

```

(End of definition for `\__fp_exp_Taylor:Nnnwn`, `\__fp_exp_Taylor_loop:www`, and `\__fp_exp_Taylor_break:Nww`.)

`\c__fp_exp_intarray` The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \dots, 9$ and $i = -1, \dots, 4$. Each value is expressed as $\simeq 10^p \times 0.m_1m_2m_3$ with three 8-digit blocks m_1, m_2, m_3 and an integer exponent p (one more than the scientific exponent), and these are stored in the integer array as four items: $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

28662 \intarray_const_from_clist:Nn \c__fp_exp_intarray
28663 {
28664     1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
28665     1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
28666     1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
28667     1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
28668     1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
28669     1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
28670     1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
28671     1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
28672     1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
28673     1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
28674     1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
28675     2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
28676     2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
28677     3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
28678     3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
28679     4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
28680     4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
28681     4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
28682     5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
28683     9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
28684     14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
28685     18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
28686     22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
28687     27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
28688     31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
28689     35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
28690     40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,

```

```

28691      44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
28692      87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
28693     131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
28694     174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
28695     218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
28696     261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
28697     305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
28698     348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
28699     391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
28700     435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
28701     869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
28702    1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
28703    1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
28704    2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
28705    2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
28706    3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
28707    3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
28708    3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
28709    4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
28710    8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
28711   13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
28712   17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
28713   21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
28714   26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
28715   30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
28716   34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
28717   39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
28718   }

```

(End of definition for \c_fp_exp_intarray.)

_fp_exp_pos_large:NnnNwn The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros).
_fp_exp_large_after:wnn The third argument is the integer part of our number, then we have the decimal part
 _fp_exp_large:NwN delimited by a _fp_sep:, and finally the exponent, in the range [0, 5]. Remove leading
 _fp_exp_intarray:w zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also
 _fp_exp_intarray_aux:w removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table,
and multiplying that to the current total. The loop is done by _fp_exp_large:NwN, whose #1 is the $\langle exponent \rangle$, #2 is the current mantissa, and #3 is the $\langle digit \rangle$. At the end, _fp_exp_large_after:wnn moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```

28719 \cs_new:Npn \_fp\_exp\_pos\_large:NnnNwn #1#2#3 #4#5\_fp\_sep: #6
28720 {
28721   \exp\_after:wN \exp\_after:wN \exp\_after:wN \_fp\_exp\_large:NwN
28722   \exp\_after:wN \exp\_after:wN \exp\_after:wN #6
28723   \exp\_after:wN \c\_fp\_one\_fixed\_tl
28724   \int\_value:w #3 #4 \exp\_stop\_f:
28725   #5 00000 \_fp\_sep:
28726 }
28727 \cs_new:Npn \_fp\_exp\_large:NwN #1#2\_fp\_sep: #3
28728 {
28729   \if\_case:w #3 ~
28730     \exp\_after:wN \_fp\_fixed\_continue:wn
28731   \else:
28732     \exp\_after:wN \_fp\_exp\_intarray:w

```

```

28733     \int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN \__fp_sep:
28734 \fi:
28735 #2\__fp_sep:
28736 {
28737     \if_meaning:w 0 #1
28738     \exp_after:wN \__fp_exp_large_after:wwn
28739     \else:
28740     \exp_after:wN \__fp_exp_large:NwN
28741     \int_value:w \__fp_int_eval:w #1 - 1 \exp_after:wN \scan_stop:
28742     \fi:
28743 }
28744 }
28745 \cs_new:Npn \__fp_exp_intarray:w #1 \__fp_sep:
28746 {
28747     +
28748     \__kernel_intarray_item:Nn \c__fp_exp_intarray
28749     { \__fp_int_eval:w #1 - 3 \scan_stop: }
28750     \exp_after:wN \use_i:nnn
28751     \exp_after:wN \__fp_fixed_mul:wwn
28752     \int_value:w 0
28753     \exp_after:wN \__fp_exp_intarray_aux:w
28754     \int_value:w \__kernel_intarray_item:Nn
28755     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
28756     \exp_after:wN \__fp_exp_intarray_aux:w
28757     \int_value:w \__kernel_intarray_item:Nn
28758     \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
28759     \exp_after:wN \__fp_exp_intarray_aux:w
28760     \int_value:w \__kernel_intarray_item:Nn
28761     \c__fp_exp_intarray {#1} \__fp_sep: \__fp_sep:
28762 }
28763 \cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 \__fp_sep:
28764 { \__fp_sep: {#1#2#3#4} {#5} }
28765 \cs_new:Npn \__fp_exp_large_after:wwn #1\__fp_sep: #2\__fp_sep: #3
28766 {
28767     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2\__fp_sep: {} #3
28768     \__fp_fixed_mul:wwn #1\__fp_sep:
28769 }

```

(End of definition for __fp_exp_pos_large:NnnNwn and others.)

81.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	$-\text{integer}$	± 0	$+\text{integer}$	$(0, \infty)$	$+\infty$	nan
$+\infty$	$+0$		$+0$	$+1$	$+\infty$	$+\infty$	$+\infty$	nan
$(1, \infty)$	$+0$		$+ a ^b$	$+1$	$+ a ^b$	$+\infty$	$+\infty$	nan
$+1$	$+1$		$+1$	$+1$	$+1$	$+1$	$+1$	$+1$
$(0, 1)$	$+\infty$		$+ a ^b$	$+1$	$+ a ^b$	$+0$	$+\infty$	nan
$+0$	$+\infty$		$+\infty$	$+1$	$+0$	$+0$	$+\infty$	nan
-0	$+\infty$	nan	$(-1)^b \infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	nan
$(-1, 0)$	$+\infty$	nan	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	nan	$+0$	nan
-1	$+1$	nan	$(-1)^b$	$+1$	$(-1)^b$	nan	$+1$	nan
$(-\infty, -1)$	$+0$	nan	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	nan	$+\infty$	nan
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b \infty$	nan	$+\infty$	nan
nan	nan	nan	nan	$+1$	nan	nan	nan	nan

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\text{nan}^0 = 1^{\text{nan}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`\_fp\_^o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a , even nan . Then test the sign of a .

- If it is positive, and a is a normal number, call `\_fp_pow_normal_o:ww` followed by the two `fp` a and b . For $a = +0$ or $+\text{inf}$, call `\_fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a nan , then skip to the next `\_fp_sep`: (which happens to be conveniently the end of b) and return nan .
- Finally, if a is negative, compute a^b (`\_fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing $\{ b a \}$, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

28770 \cs_new:cpn { \_fp\_ \iow_char:N \^ \_o:ww }
28771   \s__fp \_fp_chk:w #1#2#3\_fp_sep: \s__fp \_fp_chk:w #4#5#6\_fp_sep:
28772   {
28773     \if_meaning:w 0 #4
28774       \_fp_case_return_o:Nw \c_one_fp
28775     \fi:
28776     \if_case:w #2 \exp_stop_f:
28777       \exp_after:wN \use_i:nn
28778     \or:
28779       \_fp_case_return_o:Nw \c_nan_fp
28780     \else:
28781       \exp_after:wN \_fp_pow_neg:www
28782       \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
28783     \fi:
28784     {
28785       \if_meaning:w 1 #1
28786         \exp_after:wN \_fp_pow_normal_o:ww
28787       \else:
28788         \exp_after:wN \_fp_pow_zero_or_inf:ww
28789       \fi:
28790       \s__fp \_fp_chk:w #1#2#3\_fp_sep:
28791     }

```

```

28792     { \s__fp \__fp_chk:w #4#5#6\__fp_sep: \s__fp \__fp_chk:w #1#2#3\__fp_sep: }
28793     \s__fp \__fp_chk:w #4#5#6\__fp_sep:
28794 }

```

(End of definition for __fp_^o:ww.)

__fp_pow_zero_or_inf:ww Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```

28795 \cs_new:Npn \__fp_pow_zero_or_inf:ww
28796   \s__fp \__fp_chk:w #1#2\__fp_sep: \s__fp \__fp_chk:w #3#4
28797   {
28798     \if_meaning:w 1 #4
28799       \__fp_case_return_same_o:w
28800     \fi:
28801     \if_meaning:w #1 #4
28802       \__fp_case_return_o:Nw \c_zero_fp
28803     \fi:
28804     \if_meaning:w 2 #1
28805       \__fp_case_return_o:Nw \c_inf_fp
28806     \fi:
28807     \if_meaning:w 2 #3
28808       \__fp_case_return_o:Nw \c_inf_fp
28809     \else:
28810       \__fp_case_use:nw
28811       {
28812         \__fp_division_by_zero_o:NNww \c_inf_fp ^
28813         \s__fp \__fp_chk:w #1 #2 \__fp_sep:
28814       }
28815     \fi:
28816     \s__fp \__fp_chk:w #3#4
28817   }

```

(End of definition for __fp_pow_zero_or_inf:ww.)

__fp_pow_normal_o:ww We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1 , unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

- 0 Impossible, we already filtered $b = \pm 0$.
- 1 Call `\__fp_pow_npos_o:Nww`.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

28818 \cs_new:Npn \__fp_pow_normal_o:ww
28819   \s__fp \__fp_chk:w 1 #1#2#3\__fp_sep: \s__fp \__fp_chk:w #4#5
28820   {
28821     \if:w 0 \__fp_str_if_eq:nn { #2 #3 } { 1 {1000} {0000} {0000} {0000} }

```



```

28822     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
28823     \exp_after:wN \__fp_case_return_ii_o:ww
28824     \fi:
28825     \__fp_case_return_o:Nww \c_one_fp
28826   \fi:
28827   \if_case:w #4 \exp_stop_f:
28828   \or:
28829     \exp_after:wN \__fp_pow_npos_o:Nww
28830     \exp_after:wN #5
28831   \or:
28832     \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
28833     \if_int_compare:w #2 > \c_zero_int
28834       \exp_after:wN \__fp_case_return_o:Nww
28835       \exp_after:wN \c_inf_fp
28836     \else:
28837       \exp_after:wN \__fp_case_return_o:Nww
28838       \exp_after:wN \c_zero_fp
28839     \fi:
28840   \or:
28841     \__fp_case_return_ii_o:ww
28842   \fi:
28843   \s__fp \__fp_chk:w 1 #1 {#2} #3 \__fp_sep:
28844   \s__fp \__fp_chk:w #4 #5
28845 }

```

(End of definition for __fp_pow_normal_o:ww.)

__fp_pow_npos_o:Nww We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of __fp_ln_o:w is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

28846 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
28847 {
28848   \exp_after:wN \__fp_sanitizew:Nw
28849   \exp_after:wN 0
28850   \int_value:w
28851   \if:w #1 \if_int_compare:w #3 > \c_zero_int 0 \else: 2 \fi:
28852     \exp_after:wN \__fp_pow_npos_aux:NNww
28853     \exp_after:wN +
28854     \exp_after:wN \__fp_fixed_to_float_o:wN
28855   \else:
28856     \exp_after:wN \__fp_pow_npos_aux:NNww
28857     \exp_after:wN -
28858     \exp_after:wN \__fp_fixed_inv_to_float_o:wN
28859   \fi:
28860   {#3}
28861 }

```

(End of definition for __fp_pow_npos_o:Nww.)

__fp_pow_npos_aux:NNnw The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

28862 \cs_new:Npn \__fp_pow_npos_aux:NNnw
28863   #1#2#3#4#5\__fp_sep: \s__fp \__fp_chk:w 1#6#7#8\__fp_sep:
28864   {
28865     #1
28866     \__fp_int_eval:w
28867     \__fp_ln_significand:NNNNnnnN #4#5
28868     \__fp_pow_exponent:wnN {#3}
28869     \__fp_fixed_mul:wnN #8 {0000}{0000} \__fp_sep:
28870     \__fp_pow_B:wnN #7\__fp_sep:
28871     #1 #2 0 % fixed_to_float_o:wn
28872   }
28873 \cs_new:Npn \__fp_pow_exponent:wnN #1\__fp_sep: #2
28874   {
28875     \if_int_compare:w #2 > \c_zero_int
28876       \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
28877       \exp_after:wN +
28878     \else:
28879       \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(ln|\ln(10) + (-\ln(x)))
28880       \exp_after:wN -
28881     \fi:
28882     #2\__fp_sep: #1\__fp_sep:
28883   }
28884 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2\__fp_sep: #3#4#5#6#7#8\__fp_sep:
28885   { %^^A todo: use that in ln.
28886     \exp_after:wN \__fp_fixed_mul_after:wnn
28887     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
28888     \exp_after:wN \__fp_pack:NNNNNw
28889     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
28890     #1#2*23025 - #1 #3
28891     \exp_after:wN \__fp_pack:NNNNNw
28892     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
28893     #1 #2*8509 - #1 #4
28894     \exp_after:wN \__fp_pack:NNNNNw
28895     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
28896     #1 #2*2994 - #1 #5
28897     \exp_after:wN \__fp_pack:NNNNNw
28898     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
28899     #1 #2*0456 - #1 #6
28900     \exp_after:wN \__fp_pack:NNNNNw
28901     \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
28902     #1 #2*8401 - #1 #7
28903     #1 ( #2*7991 - #8 ) / 1 0000 \__fp_sep: \__fp_sep:
28904   }
28905 \cs_new:Npn \__fp_pow_B:wnN #1#2#3#4#5#6\__fp_sep: #7\__fp_sep:
28906   {
28907     \if_int_compare:w #7 < \c_zero_int
28908       \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
28909     \else:
28910       \if_int_compare:w #7 < 22 \exp_stop_f:
28911         \exp_after:wN \__fp_pow_C_pos:w \int_value:w
28912       \else:
28913         \exp_after:wN \__fp_pow_C_overflow:w \int_value:w

```

```

28914     \fi:
28915 \fi:
28916 #7 \exp_after:wN \__fp_sep:
28917 \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
28918 #2#3#4#5#6 0000 0000 0000 0000 0000 0000 \__fp_sep: %^A todo: how many 0?
28919 }
28920 \cs_new:Npn \__fp_pow_C_overflow:w #1\__fp_sep: #2\__fp_sep: #3
28921 {
28922   + 2 * \c__fp_max_exponent_int
28923   \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
28924 }
28925 \cs_new:Npn \__fp_pow_C_neg:w #1 \__fp_sep: 1
28926 {
28927   \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
28928   \prg_replicate:nn {#1} {0}
28929 }
28930 \cs_new:Npn \__fp_pow_C_pos:w #1\__fp_sep: 1
28931 { \__fp_pow_C_pos_loop:wN #1\__fp_sep: }
28932 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1\__fp_sep: #2
28933 {
28934   \if_meaning:w 0 #1
28935     \exp_after:wN \__fp_pow_C_pack:w
28936     \exp_after:wN #2
28937   \else:
28938     \if_meaning:w 0 #2
28939       \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
28940     \else:
28941       \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
28942     \fi:
28943     \__fp_int_eval:w #1 - 1 \exp_after:wN \__fp_sep:
28944   \fi:
28945 }
28946 \cs_new:Npn \__fp_pow_C_pack:w
28947 {
28948   \exp_after:wN \__fp_exp_large:NwN
28949   \exp_after:wN 5
28950   \c__fp_one_fixed_tl
28951 }

```

(End of definition for __fp_pow_npos_aux:NNnww.)

__fp_pow_neg:www
__fp_pow_neg_aux:wNN

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to __fp_pow_neg_aux:wNN. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or nan , in which case we return that as a^b . In particular, since the underflow detection occurs before __fp_pow_neg:www is called, $(-0.1)**(12345.67)$ gives $+0$ rather than complaining that the sign is not defined.

```

28952 \cs_new:Npn \__fp_pow_neg:www
28953   \s__fp \__fp_chk:w #1#2\__fp_sep: #3\__fp_sep: #4\__fp_sep:
28954 {
28955   \if_case:w \__fp_pow_neg_case:w #4 \__fp_sep:
28956     \exp_after:wN \__fp_pow_neg_aux:wNN
28957   \or:
28958     \if_int_compare:w \__fp_int_eval:w #1 / 2 = \c_one_int

```

```

28959         \_fp_invalid_operation_o:Nww ^ #3\_fp_sep: #4\_fp_sep:
28960         \exp:w \exp_end_continue_f:w
28961         \exp_after:wN \exp_after:wN
28962         \exp_after:wN \_fp_use_none_until_s:w
28963     \fi:
28964 \fi:
28965 \_fp_exp_after_o:w
28966 \s_fp \_fp_chk:w #1#2\_fp_sep:
28967 }
28968 \cs_new:Npn \_fp_pow_neg_aux:wNN #1 \s_fp \_fp_chk:w #2#3
28969 {
28970     \exp_after:wN \_fp_exp_after_o:w
28971     \exp_after:wN \s_fp
28972     \exp_after:wN \_fp_chk:w
28973     \exp_after:wN #2
28974     \int_value:w \_fp_int_eval:w 2 - #3 \_fp_int_eval_end:
28975 }

```

(End of definition for _fp_pow_neg:www and _fp_pow_neg_aux:wNN.)

```

\_fp_pow_neg_case:w
\_fp_pow_neg_case_aux:nnnnn
\_fp_pow_neg_case_aux:Nnnw

```

This function expects a floating point number, and determines its “parity”. It should be used after \if_case:w or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After _fp_decimate:nNnnnn the argument #1 of _fp_pow_neg_case_aux:Nnnw is a rounding digit, 0 if and only if the number was an integer, and #3 is the 8 least significant digits of that integer.

```

28976 \cs_new:Npn \_fp_pow_neg_case:w \s_fp \_fp_chk:w #1#2#3\_fp_sep:
28977 {
28978     \if_case:w #1 \exp_stop_f:
28979         -1
28980     \or: \_fp_pow_neg_case_aux:nnnnn #3
28981     \or: -1
28982     \else: 1
28983     \fi:
28984     \exp_stop_f:
28985 }
28986 \cs_new:Npn \_fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
28987 {
28988     \if_int_compare:w #1 > \c_fp_prec_int
28989         -1
28990     \else:
28991         \_fp_decimate:nNnnnn { \c_fp_prec_int - #1 }
28992         \_fp_pow_neg_case_aux:Nnnw
28993         {#2} {#3} {#4} {#5}
28994     \fi:
28995 }
28996 \cs_new:Npn \_fp_pow_neg_case_aux:Nnnw #1#2#3#4 \_fp_sep:
28997 {
28998     \if_meaning:w 0 #1
28999     \if_int_odd:w #3 \exp_stop_f:
29000         0
29001     \else:
29002         -1

```

```

29003     \fi:
29004   \else:
29005     1
29006   \fi:
29007 }

```

(End of definition for `\__fp_pow_neg_case:w`, `\__fp_pow_neg_case_aux:nnnnn`, and `\__fp_pow_neg_case_aux:Nnnw`.)

81.4 Factorial

`\c__fp_fact_max_arg_int` The maximum integer whose factorial fits in the exponent range is 3248, as $3249! \sim 10^{10000.8}$

```

29008 \int_const:Nn \c__fp_fact_max_arg_int { 3248 }

```

(End of definition for `\c__fp_fact_max_arg_int`.)

`\__fp_fact_o:w` First detect ± 0 and $+\infty$ and `nan`. Then note that factorial of anything with a negative sign (except -0) is undefined. Then call `\__fp_small_int:wTF` to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial, but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

29009 \cs_new:Npn \__fp_fact_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29010 {
29011   \if_case:w #2 \exp_stop_f:
29012     \__fp_case_return_o:Nw \c_one_fp
29013   \or:
29014   \or:
29015     \if_meaning:w 0 #3
29016     \exp_after:wN \__fp_case_return_same_o:w
29017   \fi:
29018   \or:
29019     \__fp_case_return_same_o:w
29020   \fi:
29021   \if_meaning:w 2 #3
29022     \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
29023   \fi:
29024   \__fp_fact_pos_o:w
29025   \s__fp \__fp_chk:w #2 #3 #4 \__fp_sep:
29026 }

```

(End of definition for `\__fp_fact_o:w`.)

`\__fp_fact_pos_o:w` Then check the input is an integer, and call `\__fp_factorial_int_o:n` with that `int` as an argument. If it's too big the factorial overflows. Otherwise call `\__fp_sanitize:Nw` with a positive sign marker 0 and an integer expression that will mop up any exponent in the calculation.

`\__fp_fact_int_o:w`

```

29027 \cs_new:Npn \__fp_fact_pos_o:w #1\__fp_sep:
29028 {
29029   \__fp_small_int:wTF #1\__fp_sep:
29030   { \__fp_fact_int_o:n }
29031   { \__fp_invalid_operation_o:fw { fact } #1\__fp_sep: }
29032 }

```

```

29033 \cs_new:Npn \__fp_fact_int_o:n #1
29034 {
29035   \if_int_compare:w #1 > \c__fp_fact_max_arg_int
29036     \__fp_case_return:nw
29037     {
29038       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_overflow:w
29039       \exp_after:wN \c_inf_fp
29040     }
29041   \fi:
29042   \exp_after:wN \__fp_sanitizew
29043   \exp_after:wN 0
29044   \int_value:w \__fp_int_eval:w
29045   \__fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } { } \__fp_sep:
29046 }

```

(End of definition for __fp_fact_pos_o:w and __fp_fact_int_o:w.)

__fp_fact_loop_o:w The loop receives an integer #1 whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number #2 in the form $\langle \text{exponent} \rangle, \langle \text{mantissa} \rangle \backslash\_fp\_sep:$. The loop goes in steps of two because we compute $\#1 \cdot \#1 - 1$ as an integer expression (it must fit since #1 is at most 3248), then multiply with the result so far. We don't need to fill in most of the mantissa with zeros because __fp_ep_mul:wwwwn first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than 10^8 . This limit is chosen because the normalization step cannot deal with larger integers.

```

29047 \cs_new:Npn \__fp_fact_loop_o:w #1 . #2 \__fp_sep:
29048 {
29049   \if_int_compare:w #1 < 12 \exp_stop_f:
29050     \__fp_fact_small_o:w #1
29051   \fi:
29052   \exp_after:wN \__fp_ep_mul:wwwwn
29053   \exp_after:wN 4 \exp_after:wN ,
29054   \exp_after:wN { \int_value:w \__fp_int_eval:w #1 * (#1 - 1) }
29055   { } { } { } { } { } { } \__fp_sep:
29056   #2 \__fp_sep:
29057   {
29058     \exp_after:wN \__fp_fact_loop_o:w
29059     \int_value:w \__fp_int_eval:w #1 - 2 .
29060   }
29061 }
29062 \cs_new:Npn \__fp_fact_small_o:w #1 \fi: #2 \__fp_sep: #3 \__fp_sep: #4
29063 {
29064   \fi:
29065   \exp_after:wN \__fp_ep_mul:wwwwn
29066   \exp_after:wN 4 \exp_after:wN ,
29067   \exp_after:wN
29068   {
29069     \int_value:w
29070     \if_case:w #1 \exp_stop_f:
29071     1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
29072     \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
29073     \fi:
29074     } { } { } { } { } { } { } \__fp_sep:
29075   #3 \__fp_sep:

```

```
29076     \__fp_ep_to_float_o:wwN 0
29077 }
```

(End of definition for __fp_fact_loop_o:w.)

```
29078 </code>
```

Chapter 82

l3fp-trig implementation

```
29079 (*code)
29080 (@@=fp)

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

Unary functions.
29081 \tl_map_inline:nn
29082 {
29083   {acos} {acsc} {asec} {asin}
29084   {cos} {cot} {csc} {sec} {sin} {tan}
29085 }
29086 {
29087   \cs_new:cpe { __fp_parse_word_#1:N }
29088   {
29089     \exp_not:N \__fp_parse_unary_function:NNN
29090     \exp_not:c { __fp_#1_o:w }
29091     \exp_not:N \use_i:nn
29092   }
29093   \cs_new:cpe { __fp_parse_word_#1d:N }
29094   {
29095     \exp_not:N \__fp_parse_unary_function:NNN
29096     \exp_not:c { __fp_#1_o:w }
29097     \exp_not:N \use_ii:nn
29098   }
29099 }
```

(End of definition for __fp_parse_word_acos:N and others.)

```
\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N

Those functions may receive a variable number of arguments.
29100 \cs_new:Npn \__fp_parse_word_acot:N
29101 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
29102 \cs_new:Npn \__fp_parse_word_acotd:N
29103 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
29104 \cs_new:Npn \__fp_parse_word_atan:N
29105 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
29106 \cs_new:Npn \__fp_parse_word_atand:N
29107 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
```

(End of definition for __fp_parse_word_acot:N and others.)

82.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0 , $\pm \infty$ and **nan**).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$ (in degrees, the same formula with $\pi/4 \rightarrow 45$), the sign, and the function to compute.

82.1.1 Filtering special cases

`\__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or **nan** is the same float. The sine of $\pm \infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `\__fp_sin_series_o:NNwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `\__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

29108 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29109 {
29110   \if_case:w #2 \exp_stop_f:
29111     \__fp_case_return_same_o:w
29112   \or: \__fp_case_use:nw
29113     {
29114       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
29115       \__fp_ep_to_float_o:wwN #3 0
29116     }
29117   \or: \__fp_case_use:nw
29118     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
29119   \else: \__fp_case_return_same_o:w
29120   \fi:
29121   \s__fp \__fp_chk:w #2 #3 #4\__fp_sep:
29122 }
```

(End of definition for `\__fp_sin_o:w`.)

`\__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm \infty$ raises an invalid operation exception. The cosine of **nan** is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as

for sine, but using a positive sign 0 regardless of the sign of x , and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

29123 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
29124 {
29125     \if_case:w #2 \exp_stop_f:
29126         \__fp_case_return_o:Nw \c_one_fp
29127     \or: \__fp_case_use:nw
29128         {
29129             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
29130             \__fp_ep_to_float_o:wwN 0 2
29131         }
29132     \or: \__fp_case_use:nw
29133         { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
29134     \else: \__fp_case_return_same_o:w
29135     \fi:
29136     \s__fp \__fp_chk:w #2 #3\__fp_sep:
29137 }

```

(End of definition for `\__fp_cos_o:w`.)

`\__fp_csc_o:w` The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign #3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3(\sin|x|)^{-1}$.

```

29138 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29139 {
29140     \if_case:w #2 \exp_stop_f:
29141         \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
29142     \or: \__fp_case_use:nw
29143         {
29144             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
29145             \__fp_ep_inv_to_float_o:wwN #3 0
29146         }
29147     \or: \__fp_case_use:nw
29148         { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
29149     \else: \__fp_case_return_same_o:w
29150     \fi:
29151     \s__fp \__fp_chk:w #2 #3 #4\__fp_sep:
29152 }

```

(End of definition for `\__fp_csc_o:w`.)

`\__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

29153 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
29154 {
29155     \if_case:w #2 \exp_stop_f:
29156         \__fp_case_return_o:Nw \c_one_fp

```

```

29157 \or: \__fp_case_use:nw
29158 {
29159 \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
29160 \__fp_ep_inv_to_float_o:wwN 0 2
29161 }
29162 \or: \__fp_case_use:nw
29163 { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
29164 \else: \__fp_case_return_same_o:w
29165 \fi:
29166 \s__fp \__fp_chk:w #2 #3\__fp_sep:
29167 }

```

(End of definition for __fp_sec_o:w.)

__fp_tan_o:w The tangent of ± 0 or `nan` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `\__fp_tan_series_o:NNwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `\__fp_cot_o:w` for an explanation of the 0 argument.

```

29168 \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29169 {
29170 \if_case:w #2 \exp_stop_f:
29171 \__fp_case_return_same_o:w
29172 \or: \__fp_case_use:nw
29173 {
29174 \__fp_trig:NNNNNwn #1
29175 \__fp_tan_series_o:NNwww 0 #3 1
29176 }
29177 \or: \__fp_case_use:nw
29178 { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
29179 \else: \__fp_case_return_same_o:w
29180 \fi:
29181 \s__fp \__fp_chk:w #2 #3 #4\__fp_sep:
29182 }

```

(End of definition for __fp_tan_o:w.)

__fp_cot_o:w The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw`). The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `nan` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `\__fp_tan_series_o:NNwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

29183 \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29184 {
29185 \if_case:w #2 \exp_stop_f:
29186 \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
29187 \or: \__fp_case_use:nw
29188 {
29189 \__fp_trig:NNNNNwn #1
29190 \__fp_tan_series_o:NNwww 2 #3 3
29191 }
29192 \or: \__fp_case_use:nw

```

```

29193         { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
29194     \else: \__fp_case_return_same_o:w
29195     \fi:
29196     \s__fp \__fp_chk:w #2 #3 #4\__fp_sep:
29197 }
29198 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
29199 {
29200     \fi:
29201     \token_if_eq_meaning:NNTF 0 #1
29202     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
29203     { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
29204     {#2}
29205 }

```

(End of definition for __fp_cot_o:w and __fp_cot_zero_o:Nfw.)

82.1.2 Distinguishing small and large arguments

__fp_trig:NNNNNwn

The first argument is \use_i:nn if the operand is in radians and \use_ii:nn if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`\__fp_ep_to_float_o:wN` or `\__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

29206 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8\__fp_sep:
29207 {
29208     \exp_after:wN #2
29209     \exp_after:wN #3
29210     \exp_after:wN #4
29211     \int_value:w \__fp_int_eval:w #5
29212     \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
29213     \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
29214     #1 \__fp_trig_large:ww \__fp_trigd_large:ww
29215     \else:
29216     #1 \__fp_trig_small:ww \__fp_trigd_small:ww
29217     \fi:
29218     #7,#8{0000}{0000}\__fp_sep:
29219 }

```

(End of definition for __fp_trig:NNNNNwn.)

82.1.3 Small arguments

`\__fp_trig_small:ww` This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
29220 \cs_new:Npn \__fp_trig_small:ww #1,#2\__fp_sep:
29221 { \__fp_ep_to_fixed:wwn #1,#2\__fp_sep: . #1,#2\__fp_sep: }
```

(End of definition for `\__fp_trig_small:ww`.)

`\__fp_trigd_small:ww` Convert the extended-precision number to radians, then call `\__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
29222 \cs_new:Npn \__fp_trigd_small:ww #1,#2\__fp_sep:
29223 {
29224   \__fp_ep_mul_raw:wwwN
29225   -1,{1745}{3292}{5199}{4329}{5769}{2369}\__fp_sep: #1,#2\__fp_sep:
29226   \__fp_trig_small:ww
29227 }
```

(End of definition for `\__fp_trigd_small:ww`.)

82.1.4 Argument reduction in degrees

`\__fp_trigd_large:ww`
`\__fp_trigd_large_auxi:nnnnwNNNN`
`\__fp_trigd_large_auxii:wNw`
`\__fp_trigd_large_auxiii:www`

Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle \text{block}_1 \rangle \cdots \langle \text{block}_3 \rangle . \langle \text{block}_4 \rangle \cdots \langle \text{block}_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle \text{block}_1 \rangle + \langle \text{block}_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle \text{block}_3 \rangle$. It also unpacks $\langle \text{block}_4 \rangle$ and grabs the 4 digits of $\langle \text{block}_7 \rangle$. The second auxiliary grabs the $\langle \text{block}_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle \text{block}_4 \rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new $\langle \text{block}_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `\__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle \text{blocks} \rangle$. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
29228 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7\__fp_sep:
29229 {
29230   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
29231   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
29232   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
29233   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
29234   \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
29235   \exp_after:wN \__fp_sep:
```

```

29236 \exp:w \exp_end_continue_f:w
29237 \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
29238 #2#3#4#5#6#7 0000 0000 0000 !
29239 }
29240 \cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5\__fp_sep: #6#7#8#9
29241 {
29242 \exp_after:wN \__fp_trigd_large_auxii:wNw
29243 \int_value:w \__fp_int_eval:w #1 + #2
29244 - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
29245 #3\__fp_sep:
29246 #4\__fp_sep: #5{#6#7#8#9}\__fp_sep:
29247 }
29248 \cs_new:Npn \__fp_trigd_large_auxii:wNw #1\__fp_sep: #2#3\__fp_sep:
29249 {
29250 + (#1#2 - 4) / 9 * 2
29251 \exp_after:wN \__fp_trigd_large_auxiii:www
29252 \int_value:w \__fp_int_eval:w #1#2
29253 - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 \__fp_sep:
29254 }
29255 \cs_new:Npn \__fp_trigd_large_auxiii:www #1\__fp_sep: #2\__fp_sep: #3!
29256 {
29257 \if_int_compare:w #1 < 4500 \exp_stop_f:
29258 \exp_after:wN \__fp_use_i_until_s:nw
29259 \exp_after:wN \__fp_fixed_continue:wn
29260 \else:
29261 + 1
29262 \fi:
29263 \__fp_fixed_sub:wnn {9000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29264 {#1}#2{0000}{0000}\__fp_sep:
29265 { \__fp_trigd_small:ww 2, }
29266 }

```

(End of definition for `\__fp_trigd_large:ww` and others.)

82.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are

affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`\c__fp_trig_intarray` This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is 10^8 plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 ($4 - 1$ groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```

29267 \intarray_const_from_clist:Nn \c__fp_trig_intarray
29268 {
29269     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
29270     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
29271     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
29272     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
29273     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
29274     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
29275     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
29276     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
29277     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
29278     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
29279     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
29280     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
29281     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
29282     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
29283     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
29284     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
29285     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
29286     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
29287     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
29288     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
29289     166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
29290     112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
29291     194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
29292     169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
29293     165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
29294     194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
29295     176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
29296     122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
29297     108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
29298     159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
29299     157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
29300     153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
29301     147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
29302     186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
29303     190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
29304     149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
29305     102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
29306     187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
29307     145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
29308     141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
29309     197867235, 199608024, 100273901, 108749548, 154787923, 156826113,

```

29310	199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
29311	145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
29312	193240995, 162211753, 131839402, 109707935, 170774965, 149880868,
29313	160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
29314	111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
29315	107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
29316	100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
29317	160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
29318	157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
29319	109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
29320	114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
29321	105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
29322	169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
29323	198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
29324	144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
29325	187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
29326	198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
29327	135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
29328	174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
29329	125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
29330	131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
29331	157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
29332	100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
29333	164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
29334	169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
29335	126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
29336	157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
29337	178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
29338	108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
29339	105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
29340	166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
29341	127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
29342	140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
29343	172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
29344	125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
29345	181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
29346	119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
29347	163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
29348	183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
29349	119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
29350	184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
29351	188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
29352	142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
29353	163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
29354	163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
29355	107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
29356	139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
29357	136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
29358	169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
29359	116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
29360	165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
29361	120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
29362	103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
29363	116441201, 159496011, 106328305, 120759583, 148503050, 179095584,

29364 198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
29365 198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
29366 143546572, 137557916, 113663241, 120457809, 196971566, 134022158,
29367 180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
29368 141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
29369 191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
29370 192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
29371 143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
29372 197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
29373 173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
29374 151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
29375 192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
29376 188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
29377 128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
29378 115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
29379 134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
29380 141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
29381 186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
29382 161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
29383 134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
29384 185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
29385 119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
29386 198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
29387 186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
29388 188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
29389 197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
29390 134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
29391 185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
29392 124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
29393 184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
29394 195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
29395 188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
29396 179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
29397 141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
29398 180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
29399 184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
29400 121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
29401 107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
29402 186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
29403 152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
29404 169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
29405 194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
29406 196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
29407 107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
29408 178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
29409 192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
29410 195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
29411 144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
29412 168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
29413 185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
29414 108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
29415 100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
29416 189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
29417 134324406, 157317477, 155340540, 144810061, 177612569, 108474646,

29418	114329765,	143900008,	138265211,	145210162,	136643111,	197987319,
29419	102751191,	144121361,	169620456,	193602633,	161023559,	162140467,
29420	102901215,	167964187,	135746835,	187317233,	110047459,	163339773,
29421	124770449,	118885134,	141536376,	100915375,	164267438,	145016622,
29422	113937193,	106748706,	128815954,	164819775,	119220771,	102367432,
29423	189062690,	170911791,	194127762,	112245117,	123546771,	115640433,
29424	135772061,	166615646,	174474627,	130562291,	133320309,	153340551,
29425	138417181,	194605321,	150142632,	180008795,	151813296,	175497284,
29426	167018836,	157425342,	150169942,	131069156,	134310662,	160434122,
29427	105213831,	158797111,	150754540,	163290657,	102484886,	148697402,
29428	187203725,	198692811,	149360627,	140384233,	128749423,	132178578,
29429	177507355,	171857043,	178737969,	134023369,	102911446,	196144864,
29430	197697194,	134527467,	144296030,	189437192,	154052665,	188907106,
29431	162062575,	150993037,	199766583,	167936112,	181374511,	104971506,
29432	115378374,	135795558,	167972129,	135876446,	130937572,	103221320,
29433	124605656,	161129971,	131027586,	191128460,	143251843,	143269155,
29434	129284585,	173495971,	150425653,	199302112,	118494723,	121323805,
29435	116549802,	190991967,	168151180,	122483192,	151273721,	199792134,
29436	133106764,	121874844,	126215985,	112167639,	167793529,	182985195,
29437	185453921,	106957880,	158685312,	132775454,	133229161,	198905318,
29438	190537253,	191582222,	192325972,	178133427,	181825606,	148823337,
29439	160719681,	101448145,	131983362,	137910767,	112550175,	128826351,
29440	183649210,	135725874,	110356573,	189469487,	154446940,	118175923,
29441	106093708,	128146501,	185742532,	149692127,	164624247,	183221076,
29442	154737505,	168198834,	156410354,	158027261,	125228550,	131543250,
29443	139591848,	191898263,	104987591,	115406321,	103542638,	190012837,
29444	142615518,	178773183,	175862355,	117537850,	169565995,	170028011,
29445	158412588,	170150030,	117025916,	174630208,	142412449,	112839238,
29446	105257725,	114737141,	123102301,	172563968,	130555358,	132628403,
29447	183638157,	168682846,	143304568,	105994018,	170010719,	152092970,
29448	117799058,	132164175,	179868116,	158654714,	177489647,	116547948,
29449	183121404,	131836079,	184431405,	157311793,	149677763,	173989893,
29450	102277656,	107058530,	140837477,	152640947,	143507039,	152145247,
29451	101683884,	107090870,	161471944,	137225650,	128231458,	172995869,
29452	173831689,	171268519,	139042297,	111072135,	107569780,	137262545,
29453	181410950,	138270388,	198736451,	162848201,	180468288,	120582913,
29454	153390138,	135649144,	130040157,	106509887,	192671541,	174507066,
29455	186888783,	143805558,	135011967,	145862340,	180595327,	124727843,
29456	182925939,	157715840,	136885940,	198993925,	152416883,	178793572,
29457	179679516,	154076673,	192703125,	164187609,	162190243,	104699348,
29458	159891990,	160012977,	174692145,	132970421,	167781726,	115178506,
29459	153008552,	155999794,	102099694,	155431545,	127458567,	104403686,
29460	168042864,	184045128,	181182309,	179349696,	127218364,	192935516,
29461	120298724,	169583299,	148193297,	183358034,	159023227,	105261254,
29462	121144370,	184359584,	194433836,	138388317,	175184116,	108817112,
29463	151279233,	137457721,	193398208,	119005406,	132929377,	175306906,
29464	160741530,	149976826,	147124407,	176881724,	186734216,	185881509,
29465	191334220,	175930947,	117385515,	193408089,	157124410,	163472089,
29466	131949128,	180783576,	131158294,	100549708,	191802336,	165960770,
29467	170927599,	101052702,	181508688,	197828549,	143403726,	142729262,
29468	110348701,	139928688,	153550062,	106151434,	130786653,	196085995,
29469	100587149,	139141652,	106530207,	100852656,	124074703,	166073660,
29470	153338052,	163766757,	120188394,	197277047,	122215363,	138511354,
29471	183463624,	161985542,	159938719,	133367482,	104220974,	149956672,

```

29472     170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
29473     134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
29474     168243130, 184279693, 171132070, 121049823, 123819574, 171759855,
29475     119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
29476     132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
29477     127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
29478     159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
29479     100064922, 112650013, 132686230, 121550837,
29480 }

```

(End of definition for \c__fp_trig_intarray.)

__fp_trig_large:ww The exponent #1 is between 1 and 10000. We wish to look up decimals $10^{\#1-16}/(2\pi)$ starting from the digit #1 + 1. Since they are stored in batches of 8, compute $\lfloor \#1/8 \rfloor$ and fetch blocks of 8 digits starting there. The numbering of items in \c__fp_trig_intarray starts at 1, so the block $\lfloor \#1/8 \rfloor + 1$ contains the digit we want, at one of the eight positions. Each call to \int_value:w __kernel_intarray_item:Nn expands the next, until being stopped by __fp_trig_large_auxiii:w using \exp_stop_f:. Once all these blocks are unpacked, the \exp_stop_f: and 0 to 7 digits are removed by \use_none:n...n. Finally, __fp_trig_large_auxii:w packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

29481 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6\__fp_sep:
29482 {
29483     \exp_after:wN \__fp_trig_large_auxi:w
29484     \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
29485     \int_value:w #1 , \__fp_sep:
29486     {#2}{#3}{#4}{#5} \__fp_sep:
29487 }
29488 \cs_new:Npn \__fp_trig_large_auxi:w #1, #2,
29489 {
29490     \exp_after:wN \exp_after:wN
29491     \exp_after:wN \__fp_trig_large_auxii:w
29492     \cs:w
29493         use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
29494     \exp_after:wN
29495     \cs_end:
29496     \int_value:w
29497     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29498     { \__fp_int_eval:w #1 + 1 \scan_stop: }
29499     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
29500     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29501     { \__fp_int_eval:w #1 + 2 \scan_stop: }
29502     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
29503     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29504     { \__fp_int_eval:w #1 + 3 \scan_stop: }
29505     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
29506     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29507     { \__fp_int_eval:w #1 + 4 \scan_stop: }
29508     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
29509     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29510     { \__fp_int_eval:w #1 + 5 \scan_stop: }
29511     \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
29512     \__kernel_intarray_item:Nn \c__fp_trig_intarray
29513     { \__fp_int_eval:w #1 + 6 \scan_stop: }

```

```

29514 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
29515 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
29516 { \_fp_int_eval:w #1 + 7 \scan_stop: }
29517 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
29518 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
29519 { \_fp_int_eval:w #1 + 8 \scan_stop: }
29520 \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
29521 \_kernel_intarray_item:Nn \c\_fp_trig_intarray
29522 { \_fp_int_eval:w #1 + 9 \scan_stop: }
29523 \exp_stop_f:
29524 }
29525 \cs_new:Npn \_fp_trig_large_auxii:w
29526 {
29527 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
29528 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
29529 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
29530 \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
29531 \_fp_trig_large_auxv:www \_fp_sep:
29532 }
29533 \cs_new:Npn \_fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End of definition for _fp_trig_large:ww and others.)

```

\_fp_trig_large_auxv:www
\_fp_trig_large_auxvi:wNNNNNNNN
\_fp_trig_large_pack:NNNNNw

```

First come the first 64 digits of the fractional part of $10^{1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a _fp_sep:. Then a few more digits of the same fractional part, ending with a _fp_sep:, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the auxvi auxiliary receives the significand as #2#3#4#5 and 16 digits of the fractional part as #6#7#8#9, and computes one step of the usual ladder of pack functions we use for multiplication (see e.g., _fp_fixed_mul:wwn), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last middle shift by the appropriate trailing shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last _fp_sep: closes the ladder, and we return control to the auxvii auxiliary.

```

29534 \cs_new:Npn \_fp_trig_large_auxv:www #1\_fp_sep: #2\_fp_sep: #3\_fp_sep:
29535 {
29536 \exp_after:wN \_fp_use_i_until_s:nw
29537 \exp_after:wN \_fp_trig_large_auxvii:w
29538 \int_value:w \_fp_int_eval:w \c\_fp_leading_shift_int
29539 \prg_replicate:nn { 13 }
29540 { \_fp_trig_large_auxvi:wNNNNNNNN }
29541 + \c\_fp_trailing_shift_int - \c\_fp_middle_shift_int
29542 \_fp_use_i_until_s:nw
29543 \_fp_sep: #3 #1 \_fp_sep: \_fp_sep:
29544 }
29545 \cs_new:Npn \_fp_trig_large_auxvi:wNNNNNNNN #1\_fp_sep: #2#3#4#5#6#7#8#9
29546 {
29547 \exp_after:wN \_fp_trig_large_pack:NNNNNw
29548 \int_value:w \_fp_int_eval:w \c\_fp_middle_shift_int
29549 + #2*#9 + #3*#8 + #4*#7 + #5*#6
29550 #1\_fp_sep: {#2}{#3}{#4}{#5} {#7}{#8}{#9}
29551 }

```

```

29552 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6\__fp_sep:
29553 { + #1#2#3#4#5 \__fp_sep: #6 }

```

(End of definition for __fp_trig_large_auxv:www, __fp_trig_large_auxvi:wnnnnnnnn, and __fp_trig_large_pack:NNNNNw.)

```

\__fp_trig_large_auxvii:w
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wnnnnn
\__fp_trig_large_auxxi:w

```

The `auxvii` auxiliary is followed by 52 digits and a `\__fp_sep:`. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3/125`, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `\__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

29554 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
29555 {
29556   \exp_after:wN \__fp_trig_large_auxviii:ww
29557   \int_value:w \__fp_int_eval:w (#1#2#3 - 62) / 125 \__fp_sep:
29558   #1#2#3
29559 }
29560 \cs_new:Npn \__fp_trig_large_auxviii:ww #1\__fp_sep:
29561 {
29562   + #1
29563   \if_int_odd:w #1 \exp_stop_f:
29564     \exp_after:wN \__fp_trig_large_auxix:Nw
29565     \exp_after:wN -
29566   \else:
29567     \exp_after:wN \__fp_trig_large_auxix:Nw
29568     \exp_after:wN +
29569   \fi:
29570 }
29571 \cs_new:Npn \__fp_trig_large_auxix:Nw
29572 {
29573   \exp_after:wN \__fp_use_i_until_s:nw
29574   \exp_after:wN \__fp_trig_large_auxxi:w
29575   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
29576   \prg_replicate:nn { 13 }
29577   { \__fp_trig_large_auxx:wnnnnn }
29578   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
29579   \__fp_sep:
29580 }
29581 \cs_new:Npn \__fp_trig_large_auxx:wnnnnn #1\__fp_sep: #2 #3#4#5#6
29582 {
29583   \exp_after:wN \__fp_trig_large_pack:NNNNNw
29584   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
29585   #2 8 * #3#4#5#6
29586   #1\__fp_sep: #2
29587 }
29588 \cs_new:Npn \__fp_trig_large_auxxi:w #1\__fp_sep:
29589 {
29590   \exp_after:wN \__fp_ep_mul_raw:wwwN
29591   \int_value:w \__fp_int_eval:w 0 \__fp_ep_to_ep_loop:N #1 \__fp_sep: \__fp_sep: !

```

```

29592      0,{7853}{9816}{3397}{4483}{0961}{5661}\__fp_sep:
29593      \__fp_trig_small:ww
29594    }

```

(End of definition for __fp_trig_large_auxvii:w and others.)

82.1.6 Computing the power series

__fp_sin_series_o:NNwww Here we receive a conversion function __fp_ep_to_float_o:wwN or __fp_ep_inv_to_float_o:wwN, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a __fp_sep:, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 * #4 of the argument as a fixed point number, computed with __fp_fixed_mul:wwn;
- the number itself as an extended-precision number.

If the octant is in {1,2,5,6,...}, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and __fp_sanitize:Nw checks for overflow and underflow.

```

29595 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4\__fp_sep:
29596 {
29597   \__fp_fixed_mul:wwn #4\__fp_sep: #4\__fp_sep:
29598   {
29599     \exp_after:wN \__fp_sin_series_aux_o:NNwww
29600     \exp_after:wN #1
29601     \int_value:w
29602     \if_int_odd:w \__fp_int_eval:w (#3 + 2) / 4 \__fp_int_eval_end:
29603     #2
29604     \else:
29605     \if_meaning:w #2 0 2 \else: 0 \fi:
29606     \fi:
29607     {#3}
29608   }
29609 }
29610 \cs_new:Npn \__fp_sin_series_aux_o:NNwww #1#2#3 #4\__fp_sep: #5,#6\__fp_sep:
29611 {
29612   \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
29613   \exp_after:wN \use_i:nn

```

```

29614 \else:
29615 \exp_after:wN \use_ii:nn
29616 \fi:
29617 { % 1/18!
29618 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070}\__fp_sep:
29619 #4\__fp_sep:
29620 {0000}{0000}{0000}{0477}{9477}{3324}\__fp_sep:
29621 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29622 {0000}{0000}{0011}{4707}{4559}{7730}\__fp_sep:
29623 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29624 {0000}{0000}{2087}{6756}{9878}{6810}\__fp_sep:
29625 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29626 {0000}{0027}{5573}{1922}{3985}{8907}\__fp_sep:
29627 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29628 {0000}{2480}{1587}{3015}{8730}{1587}\__fp_sep:
29629 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29630 {0013}{8888}{8888}{8888}{8888}\__fp_sep:
29631 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29632 {0416}{6666}{6666}{6666}{6666}\__fp_sep:
29633 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29634 {5000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29635 \__fp_fixed_mul_sub_back:wwwn#4\__fp_sep:
29636 {10000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29637 { \__fp_fixed_continue:wn 0, }
29638 }
29639 { % 1/17!
29640 \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254}\__fp_sep:
29641 #4\__fp_sep:
29642 {0000}{0000}{0000}{7647}{1637}{3182}\__fp_sep:
29643 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29644 {0000}{0000}{0160}{5904}{3836}{8216}\__fp_sep:
29645 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29646 {0000}{0002}{5052}{1083}{8544}{1719}\__fp_sep:
29647 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29648 {0000}{0275}{5731}{9223}{9858}{9065}\__fp_sep:
29649 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29650 {0001}{9841}{2698}{4126}{9841}{2698}\__fp_sep:
29651 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29652 {0083}{3333}{3333}{3333}{3333}\__fp_sep:
29653 \__fp_fixed_mul_sub_back:wwwn #4\__fp_sep:
29654 {1666}{6666}{6666}{6666}{6666}\__fp_sep:
29655 \__fp_fixed_mul_sub_back:wwwn#4\__fp_sep:
29656 {10000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29657 { \__fp_ep_mul:wwwn 0, } #5,#6\__fp_sep:
29658 }
29659 {
29660 \exp_after:wN \__fp_sanitizew
29661 \exp_after:wN #2
29662 \int_value:w \__fp_int_eval:w #1
29663 }
29664 #2
29665 }

```

(End of definition for __fp_sin_series_o:NNwww and __fp_sin_series_aux_o:NNwww.)

_fp_tan_series_o:NNwww
_fp_tan_series_aux_o:Nnwww

Contrarily to _fp_sin_series_o:NNwww which received a conversion auxiliary as #1, here, #1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant #3 starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor (\#3 + 1)/2 \rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first _int_value:w expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5)))}.$$

The ratio is computed by _fp_ep_div:www, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this _if_int_odd:w test relies on the fact that the octant is at least 1.

```

29666 \cs_new:Npn \_fp_tan_series_o:NNwww #1#2#3. #4\_fp_sep:
29667 {
29668   \_fp_fixed_mul:wn #4\_fp_sep: #4\_fp_sep:
29669   {
29670     \exp_after:wN \_fp_tan_series_aux_o:Nnwww
29671     \int_value:w
29672     \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
29673     \exp_after:wN \reverse_if:N
29674     \fi:
29675     \if_meaning:w #1#2 2 \else: 0 \fi:
29676     {#3}
29677   }
29678 }
29679 \cs_new:Npn \_fp_tan_series_aux_o:Nnwww #1 #2 #3\_fp_sep: #4,#5\_fp_sep:
29680 {
29681   \_fp_fixed_mul_sub_back:www {0000}{0000}{1527}{3493}{0856}{7059}\_fp_sep:
29682   #3\_fp_sep:
29683   {0000}{0159}{6080}{0274}{5257}{6472}\_fp_sep:
29684   \_fp_fixed_mul_sub_back:www #3\_fp_sep:
29685   {0002}{4571}{2320}{0157}{2558}{8481}\_fp_sep:
29686   \_fp_fixed_mul_sub_back:www #3\_fp_sep:
29687   {0115}{5830}{7533}{5397}{3168}{2147}\_fp_sep:
29688   \_fp_fixed_mul_sub_back:www #3\_fp_sep:
29689   {1929}{8245}{6140}{3508}{7719}{2982}\_fp_sep:
29690   \_fp_fixed_mul_sub_back:www #3\_fp_sep:
29691   {10000}{0000}{0000}{0000}{0000}{0000}\_fp_sep:
29692   { \_fp_ep_mul:www 0, } #4,#5\_fp_sep:
29693   {
29694     \_fp_fixed_mul_sub_back:www {0000}{0007}{0258}{0681}{9408}{4706}\_fp_sep:
29695     #3\_fp_sep:
29696     {0000}{2343}{7175}{1399}{6151}{7670}\_fp_sep:
29697     \_fp_fixed_mul_sub_back:www #3\_fp_sep:
29698     {0019}{2638}{4588}{9232}{8861}{3691}\_fp_sep:
29699     \_fp_fixed_mul_sub_back:www #3\_fp_sep:

```



```

29700                                     {0536}{6357}{0691}{4344}{6852}{4252}\__fp_sep:
29701 \__fp_fixed_mul_sub_back:wwwn #3\__fp_sep:
29702                                     {5263}{1578}{9473}{6842}{1052}{6315}\__fp_sep:
29703 \__fp_fixed_mul_sub_back:wwwn#3\__fp_sep:
29704                                     {10000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29705 {
29706     \reverse_if:N \if_int_odd:w
29707     \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
29708     \exp_after:wN \__fp_reverse_args:Nww
29709     \fi:
29710     \__fp_ep_div:wwwn 0,
29711 }
29712 }
29713 {
29714     \exp_after:wN \__fp_sanitize:Nw
29715     \exp_after:wN #1
29716     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
29717 }
29718 #1
29719 }

```

(End of definition for __fp_tan_series_o:NNwww and __fp_tan_series_aux_o:Nnwww.)

82.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, and arcsecant) are based on a function often denoted **atan2**. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of **atan** as

$$\arccos x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\arcsin x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, **atan2**, the arctangent function **atan** is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x , because $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the

result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$: otherwise replace y by $-y$ below):

- 0 $0 < |y| < 0.41421x$, then $\operatorname{atan} \frac{|y|}{x}$ is given by a nicely convergent Taylor series;
- 1 $0 < 0.41421x < |y| < x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan} \frac{x-|y|}{x+|y|}$;
- 2 $0 < 0.41421|y| < x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan} \frac{-x+|y|}{x+|y|}$;
- 3 $0 < x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan} \frac{x}{|y|}$;
- 4 $0 < -x < 0.41421|y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan} \frac{-x}{|y|}$;
- 5 $0 < 0.41421|y| < -x < |y|$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan} \frac{x+|y|}{-x+|y|}$;
- 6 $0 < -0.41421x < |y| < -x$, then $\operatorname{atan} \frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan} \frac{-x-|y|}{-x+|y|}$;
- 7 $0 < |y| < -0.41421x$, then $\operatorname{atan} \frac{|y|}{x} = \pi - \operatorname{atan} \frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

82.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw` The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result should be given in radians or in degrees. The helper `__fp_parse_function_one_two:nnw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `__fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `__fp_atan_default:w` is omitted by `__fp_parse_function_one_two:nnw`.

```

29720 \cs_new:Npn __fp_atan_o:Nw #1
29721 {
29722   __fp_parse_function_one_two:nnw
29723   { #1 { atan } { atand } }
29724   { __fp_atan_default:w __fp_atanii_o:Nww #1 }
29725 }
29726 \cs_new:Npn __fp_acot_o:Nw #1
29727 {
29728   __fp_parse_function_one_two:nnw
29729   { #1 { acot } { acotd } }
29730   { __fp_atan_default:w __fp_acotii_o:Nww #1 }
29731 }
29732 \cs_new:Npe __fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }
```

(End of definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_default:w`.)

`__fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Otherwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with

either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ± 90), or 0, leading to $\{\pm 0, \pm\pi\}$ (in degrees, $\{\pm 0, \pm 180\}$). Since $\text{acot}(x, y) = \text{atan}(y, x)$, `\_fp_acotii_o:ww` simply reverses its two arguments.

```

29733 \cs_new:Npn \_fp_atanii_o:Nww
29734   #1 \s_fp \_fp_chk:w #2#3#4\_fp_sep: \s_fp \_fp_chk:w #5 #6 @
29735   {
29736     \if_meaning:w 3 #2 \_fp_case_return_i_o:ww \fi:
29737     \if_meaning:w 3 #5 \_fp_case_return_ii_o:ww \fi:
29738     \if_case:w
29739       \if_meaning:w #2 #5
29740       \if_meaning:w 1 #2 10 \else: 0 \fi:
29741     \else:
29742       \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
29743     \fi:
29744     \exp_stop_f:
29745     \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 2 }
29746     \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 4 }
29747     \or: \_fp_case_return:nw { \_fp_atan_inf_o:NNNw #1 #3 0 }
29748     \fi:
29749     \_fp_atan_normal_o:NNnwNnw #1
29750     \s_fp \_fp_chk:w #2#3#4\_fp_sep:
29751     \s_fp \_fp_chk:w #5 #6
29752   }
29753 \cs_new:Npn \_fp_acotii_o:Nww #1#2\_fp_sep: #3\_fp_sep:
29754   { \_fp_atanii_o:Nww #1#3\_fp_sep: #2\_fp_sep: }

```

(End of definition for `\_fp_atanii_o:Nww` and `\_fp_acotii_o:Nww`.)

`\_fp_atan_inf_o:NNNw`

This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is `nan`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `\_fp_atan_combine_o:NwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\text{atan } z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\text{atan } z$ is computed to be 0, and the result is $[\#3/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

29755 \cs_new:Npn \_fp_atan_inf_o:NNNw #1#2#3 \s_fp \_fp_chk:w #4#5#6\_fp_sep:
29756   {
29757     \exp_after:wN \_fp_atan_combine_o:NwwwwN
29758     \exp_after:wN #2
29759     \int_value:w \_fp_int_eval:w
29760     \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN \_fp_sep:
29761     \c_fp_one_fixed_t1
29762     {0000}{0000}{0000}{0000}{0000}{0000}\_fp_sep:
29763     0,{0000}{0000}{0000}{0000}{0000}{0000}\_fp_sep: #1
29764   }

```

(End of definition for `\_fp_atan_inf_o:NNNw`.)

`\_fp_atan_normal_o:NNnwNnw`

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a `\_fp_sep:.` This extended precision is required by other inverse trigonometric functions, to compute things like $\text{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

29765 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
29766   #1 \s__fp \__fp_chk:w 1#2#3#4\__fp_sep: \s__fp \__fp_chk:w 1#5#6#7\__fp_sep:
29767   {
29768     \__fp_atan_test_o:NwwNwwN
29769     #2 #3, #4{0000}{0000}\__fp_sep:
29770     #5 #6, #7{0000}{0000}\__fp_sep: #1
29771   }

```

(End of definition for __fp_atan_normal_o:NNnwNnw.)

__fp_atan_test_o:NwwNwwN

This receives: the sign #1 of y , its exponent #2, its 24 digits #3 in groups of 4, and similarly for x . We prepare to call __fp_atan_combine_o:NwwwwwN which expects the sign #1, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place #1 as a first argument, and start an integer expression for the octant. The sign of x does not affect z , so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by __fp_atan_div:wnwwnw after the operands have been ordered.

```

29772 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3\__fp_sep: #4#5,#6\__fp_sep:
29773   {
29774     \exp_after:wN \__fp_atan_combine_o:NwwwwwN
29775     \exp_after:wN #1
29776     \int_value:w \__fp_int_eval:w
29777     \if_meaning:w 2 #4
29778       7 - \__fp_int_eval:w
29779     \fi:
29780     \if_int_compare:w
29781       \__fp_ep_compare:www #2,#3\__fp_sep: #5,#6\__fp_sep: > \c_zero_int
29782       3 -
29783     \exp_after:wN \__fp_reverse_args:Nww
29784     \fi:
29785     \__fp_atan_div:wnwwnw #2,#3\__fp_sep: #5,#6\__fp_sep:
29786   }

```

(End of definition for __fp_atan_test_o:NwwNwwN.)

__fp_atan_div:wnwwnw

__fp_atan_near:wwn

__fp_atan_near_aux:wwn

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

29787 \cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3\__fp_sep: #4,#5#6\__fp_sep:
29788   {
29789     \if_int_compare:w
29790       \__fp_int_eval:w 41421 * #5 < #2 000
29791       \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
29792         00 \or: 0 \fi:
29793     \exp_stop_f:

```

```

29794     \exp_after:wN \__fp_atan_near:wwwn
29795     \fi:
29796     0
29797     \__fp_ep_div:wwwn #1,{#2}#3\__fp_sep: #4,{#5}#6\__fp_sep:
29798     \__fp_atan_auxi:ww
29799   }
29800 \cs_new:Npn \__fp_atan_near:wwwn
29801   0 \__fp_ep_div:wwwn #1,#2\__fp_sep: #3,
29802   {
29803     1
29804     \__fp_ep_to_fixed:wwn #1 - #3, #2\__fp_sep:
29805     \__fp_atan_near_aux:wwn
29806   }
29807 \cs_new:Npn \__fp_atan_near_aux:wwn #1\__fp_sep: #2\__fp_sep:
29808   {
29809     \__fp_fixed_add:wwn #1\__fp_sep: #2\__fp_sep:
29810     { \__fp_fixed_sub:wwn #2\__fp_sep: #1\__fp_sep: { \__fp_ep_div:wwwn 0, } 0, }
29811   }

```

(End of definition for __fp_atan_div:wwwnw, __fp_atan_near:wwwn, and __fp_atan_near_aux:wwn.)

__fp_atan_auxi:ww Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a
 __fp_atan_auxii:w fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed
 by the fixed point representation of z and the old representation.

```

29812 \cs_new:Npn \__fp_atan_auxi:ww #1,#2\__fp_sep:
29813   { \__fp_ep_to_fixed:wwn #1,#2\__fp_sep: \__fp_atan_auxii:w #1,#2\__fp_sep: }
29814 \cs_new:Npn \__fp_atan_auxii:w #1\__fp_sep:
29815   {
29816     \__fp_fixed_mul:wwn #1\__fp_sep: #1\__fp_sep:
29817     {
29818       \__fp_atan_Taylor_loop:www 39 \__fp_sep:
29819       {0000}{0000}{0000}{0000}{0000}{0000} \__fp_sep:
29820     }
29821     ! #1\__fp_sep:
29822   }

```

(End of definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$,
 __fp_atan_Taylor_break:w $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$,
 we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then
 $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer
 expression computing the octant with a __fp_sep:, and leave the result $\#2$ afterwards.

```

29823 \cs_new:Npn \__fp_atan_Taylor_loop:www #1\__fp_sep: #2\__fp_sep: #3\__fp_sep:
29824   {
29825     \if_int_compare:w #1 = - \c_one_int
29826     \__fp_atan_Taylor_break:w
29827     \fi:
29828     \exp_after:wN \__fp_fixed_div_int:wwn \c__fp_one_fixed_t1 #1\__fp_sep:
29829     \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2\__fp_sep: #3\__fp_sep:
29830     {
29831       \exp_after:wN \__fp_atan_Taylor_loop:www
29832       \int_value:w \__fp_int_eval:w #1 - 2 \__fp_sep:
29833     }

```

```

29834     #3\__fp_sep:
29835   }
29836 \cs_new:Npn \__fp_atan_Taylor_break:w
29837   \fi: #1 \__fp_fixed_mul_sub_back:wwn #2\__fp_sep: #3 !
29838   { \fi: \__fp_sep: #2 \__fp_sep: }

```

(End of definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

```

\__fp_atan_combine_o:NwwwwN
\__fp_atan_combine_aux:ww

```

This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z , and another representation of z , as an $\langle exponent \rangle$ and the fixed point number $10^{-\langle exponent \rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to `\__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent #5 for `\__fp_sanitize:Nw`, and multiply #3 = $\frac{\operatorname{atan} z}{z}$ with #6, the adjusted z . Otherwise, multiply #3 = $\frac{\operatorname{atan} z}{z}$ with #4 = z , then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product #3 · #4. In both cases, convert to a floating point with `\__fp_fixed_to_float_o:wN`.

```

29839 \cs_new:Npn \__fp_atan_combine_o:NwwwwN
29840   #1 #2\__fp_sep: #3\__fp_sep: #4\__fp_sep: #5,#6\__fp_sep: #7
29841   {
29842     \exp_after:wN \__fp_sanitize:Nw
29843     \exp_after:wN #1
29844     \int_value:w \__fp_int_eval:w
29845     \if_meaning:w 0 #2
29846       \exp_after:wN \use_i:nn
29847     \else:
29848       \exp_after:wN \use_ii:nn
29849     \fi:
29850     { #5 \__fp_fixed_mul:wwn #3\__fp_sep: #6\__fp_sep: }
29851     {
29852       \__fp_fixed_mul:wwn #3\__fp_sep: #4\__fp_sep:
29853       {
29854         \exp_after:wN \__fp_atan_combine_aux:ww
29855         \int_value:w \__fp_int_eval:w #2 / 2 \__fp_sep: #2\__fp_sep:
29856       }
29857     }
29858     { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
29859     #1
29860   }
29861 \cs_new:Npn \__fp_atan_combine_aux:ww #1\__fp_sep: #2\__fp_sep:
29862   {
29863     \__fp_fixed_mul_short:wwn
29864     {7853}{9816}{3397}{4483}{0961}{5661}\__fp_sep:
29865     {#1}{0000}{0000}\__fp_sep:
29866     {
29867       \if_int_odd:w #2 \exp_stop_f:
29868         \exp_after:wN \__fp_fixed_sub:wwn
29869       \else:

```

```

29870         \exp_after:wN \__fp_fixed_add:wwn
29871     \fi:
29872 }
29873 }

```

(End of definition for `\__fp_atan_combine_o:NwwwwwN` and `\__fp_atan_combine_aux:ww`.)

82.2.2 Arcsine and arccosine

`\__fp_asin_o:w` Again, the first argument provided by `l3fp-parse` is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ± 0 or `nan` is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with `\__fp_acos_o:w`, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

29874 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
29875 {
29876     \if_case:w #2 \exp_stop_f:
29877         \__fp_case_return_same_o:w
29878     \or:
29879         \__fp_case_use:nw
29880         { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
29881     \or:
29882         \__fp_case_use:nw
29883         { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
29884     \else:
29885         \__fp_case_return_same_o:w
29886     \fi:
29887     \s__fp \__fp_chk:w #2 #3\__fp_sep:
29888 }

```

(End of definition for `\__fp_asin_o:w`.)

`\__fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with `\__fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```

29889 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
29890 {
29891     \if_case:w #2 \exp_stop_f:
29892         \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
29893     \or:
29894         \__fp_case_use:nw
29895         {
29896             \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
29897             \__fp_reverse_args:Nww
29898         }
29899     \or:
29900         \__fp_case_use:nw
29901         { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
29902     \else:
29903         \__fp_case_return_same_o:w
29904     \fi:
29905     \s__fp \__fp_chk:w #2 #3\__fp_sep:
29906 }

```

(End of definition for `__fp_acos_o:w`.)

`__fp_asin_normal_o:NfwNnnnnw`

If the exponent #5 is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call `__fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$, with equality only for ± 1), we also call `__fp_asin_auxi_o:NnNww`. Otherwise, `__fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

29907 \cs_new:Npn __fp_asin_normal_o:NfwNnnnnw
29908   #1#2#3 \s__fp __fp_chk:w 1#4#5#6#7#8#9\__fp_sep:
29909   {
29910     \if_int_compare:w #5 < \c_one_int
29911       \exp_after:wN __fp_use_none_until_s:w
29912     \fi:
29913     \if_int_compare:w __fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
29914       \exp_after:wN __fp_use_none_until_s:w
29915     \fi:
29916     __fp_use_i:ww
29917     __fp_invalid_operation_o:fw {#2}
29918     \s__fp __fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9}\__fp_sep:
29919     __fp_asin_auxi_o:NnNww
29920     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000}\__fp_sep:
29921   }

```

(End of definition for `__fp_asin_normal_o:NfwNnnnnw`.)

`__fp_asin_auxi_o:NnNww`

`__fp_asin_isqrt:wn`

We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x = 1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by #2 (`__fp_reverse_args:Nww` in that case) before `__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and continue after `ep_mul`.

```

29922 \cs_new:Npn __fp_asin_auxi_o:NnNww #1#2#3#4,#5\__fp_sep:
29923   {
29924     __fp_ep_to_fixed:wwn #4,#5\__fp_sep:
29925     __fp_asin_isqrt:wn
29926     __fp_ep_mul:wwwn #4,#5\__fp_sep:
29927     __fp_ep_to_ep:wwN
29928     __fp_fixed_continue:wn
29929     { #2 __fp_atan_test_o:NwwNwwN #3 }
29930     0 1,{1000}{0000}{0000}{0000}{0000}\__fp_sep: #1
29931   }
29932 \cs_new:Npn __fp_asin_isqrt:wn #1\__fp_sep:
29933   {
29934     \exp_after:wN __fp_fixed_sub:wwn \c__fp_one_fixed_tl #1\__fp_sep:
29935     {
29936       __fp_fixed_add_one:wn #1\__fp_sep:
29937       __fp_fixed_continue:wn { __fp_ep_mul:wwwn 0, } 0,

```



```

29938     }
29939     \__fp_ep_isqrt:wnn
29940 }

```

(End of definition for __fp_asin_auxi_o:NnNww and __fp_asin_isqrt:wn.)

82.2.3 Arc cosecant and arc secant

__fp_acsc_o:w Cases are mostly labeled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is $+\infty$ and 22 when the number is $-\infty$. The arc cosecant of ± 0 raises an invalid operation exception. The arc cosecant of $\pm\infty$ is ± 0 with the same sign. The arc cosecant of nan is itself. Otherwise, __fp_acsc_normal_o:NfwNnw does some more tests, keeping the function name (acsc or acscd) as an argument for invalid operation exceptions.

```

29941 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4\__fp_sep: @
29942 {
29943     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
29944         \__fp_case_use:nw
29945         { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
29946     \or: \__fp_case_use:nw
29947         { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
29948     \or: \__fp_case_return_o:Nw \c_zero_fp
29949     \or: \__fp_case_return_same_o:w
29950     \else: \__fp_case_return_o:Nw \c_minus_zero_fp
29951     \fi:
29952     \s__fp \__fp_chk:w #2 #3 #4\__fp_sep:
29953 }

```

(End of definition for __fp_acsc_o:w.)

__fp_asec_o:w The arc secant of ± 0 raises an invalid operation exception. The arc secant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arc cosecant of nan is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a __fp_reverse_args:Nww following precisely that appearing in __fp_acos_o:w.

```

29954 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3\__fp_sep: @
29955 {
29956     \if_case:w #2 \exp_stop_f:
29957         \__fp_case_use:nw
29958         { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
29959     \or:
29960         \__fp_case_use:nw
29961         {
29962             \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
29963             \__fp_reverse_args:Nww
29964         }
29965     \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
29966     \else: \__fp_case_return_same_o:w
29967     \fi:
29968     \s__fp \__fp_chk:w #2 #3\__fp_sep:
29969 }

```

(End of definition for __fp_asec_o:w.)

`__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

29970 \cs_new:Npn __fp_acsc_normal_o:NfwNnw #1#2#3 \s_fp __fp_chk:w 1#4#5#6\__fp_sep:
29971 {
29972   \int_compare:nNtF {#5} < 1
29973   {
29974     __fp_invalid_operation_o:fw {#2}
29975     \s_fp __fp_chk:w 1#4{#5}#6\__fp_sep:
29976   }
29977   {
29978     __fp_ep_div:wwwwn
29979     1,{1000}{0000}{0000}{0000}{0000}{0000}\__fp_sep:
29980     #5,#6{0000}{0000}\__fp_sep:
29981     { __fp_asin_auxi_o:NnNww #1 {#3} #4 }
29982   }
29983 }
  
```

(End of definition for `__fp_acsc_normal_o:NfwNnw`.)

```

29984 \code
  
```

Chapter 83

l3fp-convert implementation

```
29985 <*code>
29986 <@@=fp>
```

83.1 Dealing with tuples

```

    \__fp_tuple_convert:Nw
\__fp_tuple_convert_loop:nNw
    \__fp_tuple_convert_end:w

29987 \cs_new:Npn \__fp_tuple_convert:Nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 \__fp_sep:
29988 {
29989     \int_case:nnF { \__fp_array_count:n {#2} }
29990     {
29991         { 0 } { ( ) }
29992         { 1 } { \__fp_tuple_convert_end:w @ { #1 #2 , } }
29993     }
29994     {
29995         \__fp_tuple_convert_loop:nNw { } #1
29996         #2 { ? \__fp_tuple_convert_end:w } \__fp_sep:
29997         @ { \use_none:nn }
29998     }
29999 }
30000 \cs_new:Npn \__fp_tuple_convert_loop:nNw #1#2#3#4\__fp_sep: #5 @ #6
30001 {
30002     \use_none:n #3
30003     \exp_args:Nf \__fp_tuple_convert_loop:nNw { #2 #3#4 \__fp_sep: } #2 #5
30004     @ { #6 , ~ #1 }
30005 }
30006 \cs_new:Npn \__fp_tuple_convert_end:w #1 @ #2
30007 { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }
```

(End of definition for `\__fp_tuple_convert:Nw`, `\__fp_tuple_convert_loop:nNw`, and `\__fp_tuple_convert_end:w`.)

83.2 Trimming trailing zeros

```

    \__fp_trim_zeros:w
\__fp_trim_zeros_loop:w
    \__fp_trim_zeros_dot:w
    \__fp_trim_zeros_end:w
```

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing

zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

30008 \cs_new:Npn \__fp_trim_zeros:w #1 \__fp_sep:
30009 {
30010     \__fp_trim_zeros_loop:w #1 \__fp_sep:
30011     \__fp_trim_zeros_loop:w 0\__fp_sep:
30012     \__fp_trim_zeros_dot:w .\__fp_sep:
30013     \s__fp_stop
30014 }
30015 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0\__fp_sep: #2 { #2 #1 \__fp_sep: #2 }
30016 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .\__fp_sep:
30017 { \__fp_trim_zeros_end:w #1 \__fp_sep: }
30018 \cs_new:Npn \__fp_trim_zeros_end:w #1 \__fp_sep: #2 \s__fp_stop { #1 }

```

(End of definition for `\__fp_trim_zeros:w` and others.)

83.3 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `\__fp_to_scientific_dispatch:w`.

```

\fp_to_scientific:c
\fp_to_scientific:n
30019 \cs_new:Npn \fp_to_scientific:N #1
30020 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
30021 \cs_generate_variant:Nn \fp_to_scientific:N { c }
30022 \cs_new:Npn \fp_to_scientific:n
30023 {
30024     \exp_after:wN \__fp_to_scientific_dispatch:w
30025     \exp:w \exp_end_continue_f:w \__fp_parse:n
30026 }

```

(End of definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 269.)

`\__fp_to_scientific_dispatch:w` We allow tuples.

```

\__fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
30027 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
30028 {
30029     \__fp_change_func_type:NNN
30030     #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
30031     #1
30032 }
30033 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 \__fp_sep:
30034 {
30035     \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 \__fp_sep: } } { } { }
30036     nan
30037 }
30038 \cs_new:Npn \__fp_tuple_to_scientific:w
30039 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }

```

(End of definition for `\__fp_to_scientific_dispatch:w`, `\__fp_to_scientific_recover:w`, and `\__fp_tuple_to_scientific:w`.)

`\__fp_to_scientific:w` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then

filter the special cases: ± 0 are represented as 0; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; `nan` is represented as 0 after an “invalid_operation” exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

30040 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
30041 {
30042   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
30043   \if_case:w #1 \exp_stop_f:
30044     \__fp_case_return:nw { 0.000000000000000e0 }
30045   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
30046   \or:
30047     \__fp_case_use:nw
30048     {
30049       \__fp_invalid_operation:nnw
30050       { \fp_to_scientific:N \c__fp_overflowing_fp }
30051       { fp_to_scientific }
30052     }
30053   \or:
30054     \__fp_case_use:nw
30055     {
30056       \__fp_invalid_operation:nnw
30057       { \fp_to_scientific:N \c_zero_fp }
30058       { fp_to_scientific }
30059     }
30060   \fi:
30061   \s__fp \__fp_chk:w #1 #2
30062 }
30063 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
30064 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 \__fp_sep:
30065 {
30066   \exp_after:wN \__fp_to_scientific_normal:wNw
30067   \exp_after:wN e
30068   \int_value:w \__fp_int_eval:w #2 - 1
30069   \__fp_sep: #3 #4 #5 #6 \__fp_sep:
30070 }
30071 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 \__fp_sep: #2#3\__fp_sep:
30072 { #2.#3 #1 }

```

(End of definition for `\__fp_to_scientific:w`, `\__fp_to_scientific_normal:wnnnnn`, and `\__fp_to_scientific_normal:wNw`.)

83.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `\__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```

\fp_to_decimal:c
\fp_to_decimal:n
30073 \cs_new:Npn \fp_to_decimal:N #1
30074 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
30075 \cs_generate_variant:Nn \fp_to_decimal:N { c }
30076 \cs_new:Npn \fp_to_decimal:n
30077 {
30078   \exp_after:wN \__fp_to_decimal_dispatch:w
30079   \exp:w \exp_end_continue_f:w \__fp_parse:n

```

```
30080 }
```

(End of definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 268.)

```
\__fp_to_decimal_dispatch:w
\__fp_to_decimal_recover:w
\__fp_tuple_to_decimal:w
```

We allow tuples.

```
30081 \cs_new:Npn \__fp_to_decimal_dispatch:w #1
30082 {
30083   \__fp_change_func_type:NNN
30084   #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
30085   #1
30086 }
30087 \cs_new:Npn \__fp_to_decimal_recover:w #1 #2 \__fp_sep:
30088 {
30089   \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 \__fp_sep: } } { } { }
30090   nan
30091 }
30092 \cs_new:Npn \__fp_tuple_to_decimal:w
30093 { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }
```

(End of definition for `\__fp_to_decimal_dispatch:w`, `\__fp_to_decimal_recover:w`, and `\__fp_tuple_to_decimal:w`.)

```
\__fp_to_decimal:w
\__fp_to_decimal_normal:wnnnnn
\__fp_to_decimal_large:Nnnw
\__fp_to_decimal_huge:wnnnn
```

The structure is similar to `\__fp_to_scientific:w`. Insert `-` for negative numbers. Zero gives 0, $\pm\infty$ and `nan` yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range $[1, 15]$ have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be `0.<zeros><digits>`, trimmed.

```
30094 \cs_new:Npn \__fp_to_decimal:w \s_fp \__fp_chk:w #1#2
30095 {
30096   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
30097   \if_case:w #1 \exp_stop_f:
30098     \__fp_case_return:nw { 0 }
30099   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
30100   \or:
30101     \__fp_case_use:nw
30102     {
30103       \__fp_invalid_operation:nnw
30104       { \fp_to_decimal:N \c__fp_overflowing_fp }
30105       { fp_to_decimal }
30106     }
30107   \or:
30108     \__fp_case_use:nw
30109     {
30110       \__fp_invalid_operation:nnw
30111       { 0 }
30112       { fp_to_decimal }
30113     }
30114   \fi:
30115   \s_fp \__fp_chk:w #1 #2
30116 }
```

```

30117 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
30118   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 \__fp_sep:
30119   {
30120     \int_compare:nNnTF {#2} > 0
30121     {
30122       \int_compare:nNnTF {#2} < \c__fp_prec_int
30123       {
30124         \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
30125         \__fp_to_decimal_large:Nnnw
30126       }
30127       {
30128         \exp_after:wN \exp_after:wN
30129         \exp_after:wN \__fp_to_decimal_huge:wnnnn
30130         \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } \__fp_sep:
30131       }
30132       {#3} {#4} {#5} {#6}
30133     }
30134     {
30135       \exp_after:wN \__fp_trim_zeros:w
30136       \exp_after:wN 0
30137       \exp_after:wN .
30138       \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
30139       #3#4#5#6 \__fp_sep:
30140     }
30141   }
30142 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4\__fp_sep:
30143   {
30144     \exp_after:wN \__fp_trim_zeros:w \int_value:w
30145     \if_int_compare:w #2 > \c_zero_int
30146       #2
30147     \fi:
30148     \exp_stop_f:
30149     #3.#4 \__fp_sep:
30150   }
30151 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1\__fp_sep: #2#3#4#5 { #2#3#4#5 #1 }

```

(End of definition for __fp_to_decimal:w and others.)

83.5 Token list representation

\fp_to_tl:N These three public functions evaluate their argument, then pass it to __fp_to_tl_dispatch:w.
\fp_to_tl:c
\fp_to_tl:n

```

30152 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
30153 \cs_generate_variant:Nn \fp_to_tl:N { c }
30154 \cs_new:Npn \fp_to_tl:n
30155   {
30156     \exp_after:wN \__fp_to_tl_dispatch:w
30157     \exp:w \exp_end_continue_f:w \__fp_parse:n
30158   }

```

(End of definition for \fp_to_tl:N and \fp_to_tl:n. These functions are documented on page 269.)

```

\__fp_to_tl_dispatch:w
\__fp_to_tl_recover:w
\__fp_tuple_to_tl:w

```

We allow tuples.

```

30159 \cs_new:Npn \__fp_to_tl_dispatch:w #1
30160 { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
30161 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 \__fp_sep:
30162 {
30163   \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 \__fp_sep: } } { } { }
30164   nan
30165 }
30166 \cs_new:Npn \__fp_tuple_to_tl:w
30167 { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

```

(End of definition for __fp_to_tl_dispatch:w, __fp_to_tl_recover:w, and __fp_tuple_to_tl:w.)

__fp_to_tl:w A structure similar to __fp_to_scientific_dispatch:w and __fp_to_decimal_dispatch:w, but without the “invalid operation” exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```

30168 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
30169 {
30170   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
30171   \if_case:w #1 \exp_stop_f:
30172     \__fp_case_return:nw { 0 }
30173   \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
30174   \or: \__fp_case_return:nw { inf }
30175   \else: \__fp_case_return:nw { nan }
30176   \fi:
30177 }
30178 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
30179 {
30180   \int_compare:nTF
30181     { -2 <= #1 <= \c__fp_prec_int }
30182     { \__fp_to_decimal_normal:wnnnnn }
30183     { \__fp_to_tl_scientific:wnnnnn }
30184   \s__fp \__fp_chk:w 1 0 {#1}
30185 }
30186 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
30187 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 \__fp_sep:
30188 {
30189   \exp_after:wN \__fp_to_tl_scientific:wNw
30190   \exp_after:wN e
30191   \int_value:w \__fp_int_eval:w #2 - 1
30192   \__fp_sep: #3 #4 #5 #6 \__fp_sep:
30193 }
30194 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 \__fp_sep: #2#3\__fp_sep:
30195 { \__fp_trim_zeros:w #2.#3 \__fp_sep: #1 }

```

(End of definition for __fp_to_tl:w and others.)

83.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

83.7 Convert to dimension or integer

\fp_to_dim:N All three public variants are based on the same `\__fp_to_dim_dispatch:w` after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

```

\__fp_to_dim_dispatch:w 30196 \cs_new:Npn \fp_to_dim:N #1
\__fp_to_dim_recover:w 30197 { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
\__fp_to_dim:w 30198 \cs_generate_variant:Nn \fp_to_dim:N { c }
30199 \cs_new:Npn \fp_to_dim:n
30200 {
30201   \exp_after:wN \__fp_to_dim_dispatch:w
30202   \exp:w \exp_end_continue_f:w \__fp_parse:n
30203 }
30204 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 \__fp_sep:
30205 {
30206   \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
30207   #1 #2 \__fp_sep:
30208 }
30209 \cs_new:Npn \__fp_to_dim_recover:w #1
30210 { \__fp_invalid_operation:nw { 0pt } { fp_to_dim } }
30211 \cs_new:Npn \__fp_to_dim:w #1 \__fp_sep: { \__fp_to_decimal:w #1 \__fp_sep: pt }
```

(End of definition for `\fp_to_dim:N` and others. These functions are documented on page 268.)

\fp_to_int:N For the most part identical to `\fp_to_dim:N` but without `pt`, and where `\__fp_to_int:w` does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `\__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```

\__fp_to_int_dispatch:w 30212 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
\__fp_to_int_recover:w 30213 \cs_generate_variant:Nn \fp_to_int:N { c }
30214 \cs_new:Npn \fp_to_int:n
30215 {
30216   \exp_after:wN \__fp_to_int_dispatch:w
30217   \exp:w \exp_end_continue_f:w \__fp_parse:n
30218 }
30219 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 \__fp_sep:
30220 {
30221   \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
30222   #1 #2 \__fp_sep:
30223 }
30224 \cs_new:Npn \__fp_to_int_recover:w #1
30225 { \__fp_invalid_operation:nw { 0 } { fp_to_int } }
30226 \cs_new:Npn \__fp_to_int:w #1\__fp_sep:
30227 {
30228   \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
30229   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1\__fp_sep: { 0 }
30230 }
```

(End of definition for `\fp_to_int:N` and others. These functions are documented on page 268.)

83.8 Convert from a dimension

\dim_to_fp:n The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (i.e., expressed in scaled points), then multiplied by $2^{-16} =$

```

\__fp_from_dim_test:w
\__fp_from_dim:wNw
\__fp_from_dim:wNNnnnnnn
\__fp_from_dim:wnnnnwNw
```

0.0000152587890625 to give a value expressed in points. The auxiliary `\__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s__fp ... \__fp_sep:`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `\__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```

30231 \cs_new:Npn \dim_to_fp:n #1
30232 {
30233   \exp_after:wN \__fp_from_dim_test:ww
30234   \exp_after:wN 0
30235   \exp_after:wN ,
30236   \int_value:w \tex_glueexpr:D #1 \__fp_sep:
30237 }
30238 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
30239 {
30240   \if_meaning:w 0 #2
30241     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
30242   \else:
30243     \exp_after:wN \__fp_from_dim:wNw
30244     \int_value:w \__fp_int_eval:w #1 - 4
30245     \if_meaning:w - #2
30246       \exp_after:wN , \exp_after:wN 2 \int_value:w
30247     \else:
30248       \exp_after:wN , \exp_after:wN 0 \int_value:w #2
30249     \fi:
30250   \fi:
30251 }
30252 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3\__fp_sep:
30253 {
30254   \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn \__fp_sep:
30255   #3 000 0000 00 {10}987654321\__fp_sep: #2 {#1}
30256 }
30257 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1\__fp_sep: #2#3#4#5#6#7#8#9
30258 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} \__fp_sep: }
30259 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1\__fp_sep: #2#3#4#5#6\__fp_sep: #7#8
30260 {
30261   \__fp_mul_npos_o:Nww #7
30262   \s__fp \__fp_chk:w 1 #7 {#5} #1 \__fp_sep:
30263   \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} \__fp_sep:
30264   \prg_do_nothing:
30265 }

```

(End of definition for `\dim_to_fp:n` and others. This function is documented on page 237.)

83.9 Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.
`\fp_use:c` 30266 `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`
`\fp_eval:n` 30267 `\cs_generate_variant:Nn \fp_use:N { c }`
30268 `\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n`

(End of definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 269.)

\fp_sign:n Trivial but useful. See the implementation of \fp_add:Nn for an explanation of why to use __fp_parse:n, namely, for better error reporting.

```
30269 \cs_new:Npn \fp_sign:n #1
30270 { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }
```

(End of definition for \fp_sign:n. This function is documented on page 268.)

\fp_abs:n Trivial but useful. See the implementation of \fp_add:Nn for an explanation of why to use __fp_parse:n, namely, for better error reporting.

```
30271 \cs_new:Npn \fp_abs:n #1
30272 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(End of definition for \fp_abs:n. This function is documented on page 287.)

\fp_max:nn Similar to \fp_abs:n, for consistency with \int_max:nn, etc.

\fp_min:nn

```
30273 \cs_new:Npn \fp_max:nn #1#2
30274 { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
30275 \cs_new:Npn \fp_min:nn #1#2
30276 { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(End of definition for \fp_max:nn and \fp_min:nn. These functions are documented on page 287.)

83.10 Convert an array of floating points to a comma list

__fp_array_to_clist:n Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The \use_ii:nn function is expanded after __fp_expand:n is done, and it removes ,~ from the start of the representation.

```
30277 \cs_new:Npn \__fp_array_to_clist:n #1
30278 {
30279   \tl_if_empty:nF {#1}
30280   {
30281     \exp_last_unbraced:Ne \use_ii:nn
30282     {
30283       \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } \__fp_sep:
30284       \prg_break_point:
30285     }
30286   }
30287 }
30288 \cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2\__fp_sep:
30289 {
30290   \use_none:n #1
30291   , ~
```

```

30292     \exp_not:f { \__fp_to_tl_dispatch:w #1 #2 \__fp_sep: }
30293     \__fp_array_to_clist_loop:Nw
30294   }

(End of definition for \__fp_array_to_clist:n and \__fp_array_to_clist_loop:Nw.)

30295 \</code>

```

Chapter 84

l3fp-random implementation

```
30296 <*code>
30297 <@@=fp>

\__fp_parse_word_rand:N Those functions may receive a variable number of arguments. We won't use the argu-
\__fp_parse_word_randint:N ment ?.

30298 \cs_new:Npn \__fp_parse_word_rand:N
30299 { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
30300 \cs_new:Npn \__fp_parse_word_randint:N
30301 { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }

(End of definition for \__fp_parse_word_rand:N and \__fp_parse_word_randint:N.)
```

84.1 Engine support

Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo 2^{28} . When `\tex_uniformdeviate:D` (`<integer>`) is called (for brevity denote by N the `<integer>`), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and N from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to $N-1$ if N is a divisor of 2^{28} , so we will mostly call the RNG with such power of 2 arguments. If N does not divide 2^{28} , then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value X requires about $2^{56}/N$ random trials. But collective patterns can reduce this to about $2^{56}/N^2$. For instance with $N = 3 \times 2^k$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^k/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly 2^{26} calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as N is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo 2^{28} , hence the lowest k bits of the random numbers only depend on the lowest k bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N - 1$ is thus to scale the raw 28-bit integer, as the engine's RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument N to get a set of K integers in $[0, N - 1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N - 1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo N (a more detailed analysis shows that 0 appears with frequency close to $3/4$). The resulting set thus has more triplets (a, b, c) than expected obeying $a = b + c$ modulo N . Namely it will have of order $(K - 55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the x_{2n} on the one hand and between the x_{2n+1} on the other hand. Such relations will have more complicated coefficients than ± 1 , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument 2^{28} or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.
- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in $\text{\TeX}$, so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behavior must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int_rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int_rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to $2 \times 10^{16} - 1$ possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by `random(N)` one call to `\tex_uniformdeviate:D` with argument N , and by `ediv(p, q)` the ε - $\text{\TeX}$ rounding division giving $\lfloor p/q + 1/2 \rfloor$. Denote by $\langle \min \rangle$, $\langle \max \rangle$

and $R = \langle \text{max} \rangle - \langle \text{min} \rangle + 1$ the arguments of `\int_min:nn` and the number of possible outcomes. Note that $R \in [1, 2^{32} - 1]$ cannot necessarily be represented as an integer (however, $R - 2^{31}$ can). Our strategy is to get two 28-bit integers X and Y from the RNG, split each into 14-bit integers, as $X = X_1 \times 2^{14} + X_0$ and $Y = Y_1 \times 2^{14} + Y_0$ then return essentially $\langle \text{min} \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$. For small R the X_0 term has a tiny effect so we ignore it and we can compute $R \times Y/2^{28}$ much more directly by `random(R)`.

- If $R \leq 2^{17} - 1$ then return $\text{ediv}(R \text{ random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \text{min} \rangle$. The shifts by 2^{13} and -1 convert $\varepsilon\text{-TeX}$ division to truncated division. The bound on R ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order $2^{17}/2^{42} = 2^{-25}$.
- Split $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$, where $R_2 \in [0, 15]$. Compute $\langle \text{min} \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$ then map a result of $\langle \text{max} \rangle + 1$ to $\langle \text{min} \rangle$. Writing each `ediv` in terms of truncated division with a shift, and using $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$, what we compute is equal to $\lfloor \langle \text{exact} \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$ with $\langle \text{exact} \rangle = \langle \text{min} \rangle + R \times 0.X_1 Y_1 Y_0 X_0$. Given we map $\langle \text{max} \rangle + 1$ to $\langle \text{min} \rangle$, the shift has no effect on uniformity. The non-uniformity is bounded by $R/2^{56} < 2^{-24}$. It may be possible to speed up the code by dropping tiny terms such as $R_0 X_0$, but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields $\langle \text{max} \rangle + 1$ with $\langle \text{max} \rangle = 2^{31} - 1$ (note that R is then arbitrary), we compute the result in two pieces. Compute $\langle \text{first} \rangle = \langle \text{min} \rangle + R_2 X_1 2^{14}$ if $R_2 < 8$ or $\langle \text{min} \rangle + 8 X_1 2^{14} + (R_2 - 8) X_1 2^{14}$ if $R_2 \geq 8$, the expressions being chosen to avoid overflow. Compute $\langle \text{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$, at most $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$, not at risk of overflowing. We have $\langle \text{first} \rangle + \langle \text{second} \rangle = \langle \text{max} \rangle + 1 = \langle \text{min} \rangle + R$ if and only if $\langle \text{second} \rangle = R 2^{14} + R_0 + R_2 2^{14}$ and $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$ (namely $R_2 = 0$ or $X_1 = 2^{14} - 1$). In that case, return $\langle \text{min} \rangle$, otherwise return $\langle \text{first} \rangle + \langle \text{second} \rangle$, which is safe because it is at most $\langle \text{max} \rangle$. Note that the decision of what to return does not need $\langle \text{first} \rangle$ explicitly so we don't actually compute it, just put it in an integer expression in which $\langle \text{second} \rangle$ is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by R and add $\langle \text{min} \rangle$. This requires some care because `l3fp-extended` only supports non-negative numbers.

`\c__kernel_randint_max_int` Constant equal to $2^{17} - 1$, the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

30302 `\int_const:Nn \c__kernel_randint_max_int { 131071 }`

(End of definition for `\c__kernel_randint_max_int`.)

`\__kernel_randint:n` Used in an integer expression, `\__kernel_randint:n {R}` gives a random number $1 + \lfloor (R \text{ random}(2^{14}) + \text{random}(R))/2^{14} \rfloor$ that is in $[1, R]$. Previous code was computing $\lfloor p/2^{14} \rfloor$ as $\text{ediv}(p - 2^{13}, 2^{14})$ but that wrongly gives -1 for $p = 0$.

```

30303 \cs_new:Npn \__kernel_randint:n #1
30304 {
30305   (#1 * \tex_uniformdeviate:D 16384
30306    + \tex_uniformdeviate:D #1 + 8192 ) / 16384
30307 }

```

(End of definition for __kernel_randint:n.)

__fp_rand_myriads:n Used as __fp_rand_myriads:n {XXX} with one letter X (specifically) per block of four digit we want; it expands to __fp_sep: followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in [10000, 19999] for the usual reason of preserving leading zeros.

```

30308 \cs_new:Npn \__fp_rand_myriads:n #1
30309 { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: \__fp_sep: }
30310 \cs_new:Npn \__fp_rand_myriads_loop:w #1 X
30311 {
30312   #1
30313   \exp_after:wN \__fp_rand_myriads_get:w
30314   \int_value:w \__fp_int_eval:w 9999 +
30315   \__kernel_randint:n { 10000 }
30316   \__fp_rand_myriads_loop:w
30317 }
30318 \cs_new:Npn \__fp_rand_myriads_get:w 1 #1 \__fp_sep: { \__fp_sep: {#1} }

```

(End of definition for __fp_rand_myriads:n, __fp_rand_myriads_loop:w, and __fp_rand_myriads_get:w.)

84.2 Random floating point

__fp_rand_o:Nw First we check that random was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the underflow flag; it should not do that.

```

30319 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
30320 {
30321   \tl_if_empty:nTF {#1}
30322   {
30323     \exp_after:wN \__fp_rand_o:w
30324     \exp:w \exp_end_continue_f:w
30325     \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } \__fp_sep: 0
30326   }
30327   {
30328     \__fp_error_num_args:ffff { rand } { 0 } { 0 }
30329     { \__fp_array_count:n {#1} }
30330     \exp_after:wN \c_nan_fp
30331   }
30332 }
30333 \cs_new:Npn \__fp_rand_o:w \__fp_sep:
30334 {
30335   \exp_after:wN \__fp_sanitize:Nw
30336   \exp_after:wN 0
30337   \int_value:w \__fp_int_eval:w \c_zero_int
30338   \__fp_fixed_to_float_o:wN
30339 }

```


(End of definition for `__fp_rand_o:Nw` and `__fp_rand_o:w`.)

84.3 Random integer

```

__fp_randint_o:Nw
__fp_randint_default:w
__fp_randint_badarg:w
__fp_randint_o:w
__fp_randint_auxi_o:ww
__fp_randint_auxii:wn
__fp_randint_auxiii_o:ww
__fp_randint_auxiv_o:ww
__fp_randint_auxv_o:w

```

Enforce that there is one argument (then add first argument 1) or two arguments. Call `__fp_randint_badarg:w` on each; this function inserts 1 `\exp_stop_f:` to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is $\geq 10^{16}$. Also bail out if `__fp_compare_back:ww` yields 1, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times 10^{-16} (hence the shift in exponent) to a 24-digit fixed point number (see `l3fp-extended`). Then compute the number of choices, $\langle max \rangle + 1 - \langle min \rangle$. Create a random 24-digit fixed-point number with `__fp_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to $\langle min \rangle$. Then truncate to 16 digits (namely select the integer part of 10^{16} times the result) before converting back to a floating point number (`__fp_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely 10^{16} to the integers they represent), except of course when it is time to convert back to a float.

```

30340 \cs_new:Npn __fp_randint_o:Nw ?
30341 {
30342   __fp_parse_function_one_two:nnw
30343   { randint }
30344   { __fp_randint_default:w __fp_randint_o:w }
30345 }
30346 \cs_new:Npn __fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
30347 \cs_new:Npn __fp_randint_badarg:w \s__fp __fp_chk:w #1#2#3\__fp_sep:
30348 {
30349   __fp_int:wTF \s__fp __fp_chk:w #1#2#3\__fp_sep:
30350   {
30351     \if_meaning:w 1 #1
30352     \if_int_compare:w
30353       __fp_use_i_until_s:nw #3 \__fp_sep: > \c__fp_prec_int
30354       \c_one_int
30355     \fi:
30356     \fi:
30357   }
30358   { \c_one_int }
30359 }
30360 \cs_new:Npn __fp_randint_o:w #1\__fp_sep: #2\__fp_sep: @
30361 {
30362   \if_case:w
30363     __fp_randint_badarg:w #1\__fp_sep:
30364     __fp_randint_badarg:w #2\__fp_sep:
30365     \if:w 1 __fp_compare_back:ww #2\__fp_sep: #1\__fp_sep: \c_one_int \fi:
30366     \c_zero_int
30367     __fp_randint_auxi_o:ww #1\__fp_sep: #2\__fp_sep:
30368   \or:
30369     __fp_invalid_operation_tl_o:ff
30370     { randint } { __fp_array_to_clist:n { #1\__fp_sep: #2\__fp_sep: } }
30371   \exp:w
30372   \fi:
30373   \exp_after:wN \exp_end:
30374 }

```

```

30375 \cs_new:Npn \__fp_randint_auxi_o:ww #1 \__fp_sep: #2 \__fp_sep: #3 \exp_end:
30376 {
30377   \fi:
30378   \__fp_randint_auxii:wn #2 \__fp_sep:
30379   { \__fp_randint_auxii:wn #1 \__fp_sep: \__fp_randint_auxiii_o:ww }
30380 }
30381 \cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1#2#3#4 \__fp_sep:
30382 {
30383   \if_meaning:w 0 #1
30384     \exp_after:wN \use_i:nn
30385   \else:
30386     \exp_after:wN \use_ii:nn
30387   \fi:
30388   { \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl }
30389   {
30390     \exp_after:wN \__fp_ep_to_fixed:wwn
30391     \int_value:w \__fp_int_eval:w
30392     #3 - \c__fp_prec_int , #4 {0000} {0000} \__fp_sep:
30393     {
30394       \if_meaning:w 0 #2
30395         \exp_after:wN \use_i:nnnn
30396         \exp_after:wN \__fp_fixed_add_one:wn
30397       \fi:
30398       \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
30399     }
30400     \__fp_fixed_continue:wn
30401   }
30402 }
30403 \cs_new:Npn \__fp_randint_auxiii_o:ww #1 \__fp_sep: #2 \__fp_sep:
30404 {
30405   \__fp_fixed_add:wwn #2 \__fp_sep:
30406   {0000} {0000} {0000} {0001} {0000} {0000} \__fp_sep:
30407   \__fp_fixed_sub:wwn #1 \__fp_sep:
30408   {
30409     \exp_after:wN \use_i:nn
30410     \exp_after:wN \__fp_fixed_mul_add:wwwn
30411     \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } \__fp_sep:
30412   }
30413   #1 \__fp_sep:
30414   \__fp_randint_auxiv_o:ww
30415   #2 \__fp_sep:
30416   \__fp_randint_auxv_o:w #1 \__fp_sep: @
30417 }
30418 \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 \__fp_sep: #6#7#8#9
30419 {
30420   \if_int_compare:w
30421     \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
30422       \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
30423     #3#4 > #8#9 \exp_stop_f:
30424     \__fp_use_i_until_s:nw
30425   \fi:
30426   \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
30427 }
30428 \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 \__fp_sep: #6 @

```

```

30429 {
30430     \exp_after:wN \__fp_sanitiz:Nw
30431     \int_value:w
30432     \if_int_compare:w #1 < 10000 \exp_stop_f:
30433         2
30434     \else:
30435         0
30436         \exp_after:wN \exp_after:wN
30437         \exp_after:wN \__fp_reverse_args:Nww
30438     \fi:
30439     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
30440     {#1} {#2} {#3} {#4} {0000} {0000} \__fp_sep:
30441     {
30442         \exp_after:wN \exp_stop_f:
30443         \int_value:w \__fp_int_eval:w \c__fp_prec_int
30444         \__fp_fixed_to_float_o:wN
30445     }
30446     0
30447     \exp:w \exp_after:wN \exp_end:
30448 }

```

(End of definition for __fp_randint_o:Nw and others.)

\int_rand:nn Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than \c__kernel_randint_max_int; #2-#1 may overflow for very large positive #2 and negative #1. If the range is narrow, call __kernel_randint:n {<choices>} where <choices> is the number of possible outcomes. If the range is wide, use somewhat slower code.

__fp_randint:ww

```

30449 \cs_new:Npn \int_rand:nn #1#2
30450 {
30451     \int_eval:n
30452     {
30453         \exp_after:wN \__fp_randint:ww
30454         \int_value:w \int_eval:n {#1} \exp_after:wN \__fp_sep:
30455         \int_value:w \int_eval:n {#2} \__fp_sep:
30456     }
30457 }
30458 \cs_new:Npn \__fp_randint:ww #1\__fp_sep: #2\__fp_sep:
30459 {
30460     \if_int_compare:w #1 > #2 \exp_stop_f:
30461     \msg_expandable_error:nnnn
30462     { kernel } { randint-backward-range } {#1} {#2}
30463     \__fp_randint:ww #2\__fp_sep: #1\__fp_sep:
30464     \else:
30465     \if_int_compare:w \__fp_int_eval:w #2
30466     \if_int_compare:w #1 > \c_zero_int
30467     - #1 < \__fp_int_eval:w
30468     \else:
30469     < \__fp_int_eval:w #1 +
30470     \fi:
30471     \c__kernel_randint_max_int
30472     \__fp_int_eval_end:
30473     \__kernel_randint:n
30474     { \__fp_int_eval:w #2 - #1 + 1 \__fp_int_eval_end: }

```

```

30475         - 1 + #1
30476     \else:
30477         \__kernel_randint:nn {#1} {#2}
30478     \fi:
30479 \fi:
30480 }

```

(End of definition for \int_rand:nn and __fp_randint:ww. This function is documented on page 183.)

Any $n \in [-2^{31} + 1, 2^{31} - 1]$ is uniquely written as $2^{14}n_1 + n_2$ with $n_1 \in [-2^{17}, 2^{17} - 1]$ and $n_2 \in [0, 2^{14} - 1]$. Calling __fp_randint_split_o:Nw n __fp_sep: gives n_1 __fp_sep: n_2 __fp_sep: and expands the next token once. We do this for two random numbers and apply __fp_randint_split_o:Nw twice to fully decompose the range R . One subtlety is that we compute $R - 2^{31} = \langle \max \rangle - \langle \min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$ rather than R to avoid overflow.

Then we have __fp_randint_wide_aux:w $\langle X_1 \rangle$ __fp_sep: $\langle X_0 \rangle$ __fp_sep: $\langle Y_1 \rangle$ __fp_sep: $\langle Y_0 \rangle$ __fp_sep: $\langle R_2 \rangle$ __fp_sep: $\langle R_1 \rangle$ __fp_sep: $\langle R_0 \rangle$ __fp_sep: . and we apply the algorithm described earlier.

```

30481 \cs_new:Npn \__kernel_randint:nn #1#2
30482 {
30483     #1
30484     \exp_after:wN \__fp_randint_wide_aux:w
30485     \int_value:w
30486     \exp_after:wN \__fp_randint_split_o:Nw
30487     \tex_uniformdeviate:D 268435456 \__fp_sep:
30488     \int_value:w
30489     \exp_after:wN \__fp_randint_split_o:Nw
30490     \tex_uniformdeviate:D 268435456 \__fp_sep:
30491     \int_value:w
30492     \exp_after:wN \__fp_randint_split_o:Nw
30493     \int_value:w \__fp_int_eval:w 131072 +
30494     \exp_after:wN \__fp_randint_split_o:Nw
30495     \int_value:w
30496     \__kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } \__fp_sep:
30497     .
30498 }
30499 \cs_new:Npn \__fp_randint_split_o:Nw #1#2 \__fp_sep:
30500 {
30501     \if_meaning:w 0 #1
30502     0 \exp_after:wN \__fp_sep: \int_value:w 0
30503     \else:
30504     \exp_after:wN \__fp_randint_split_aux:w
30505     \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 \__fp_sep:
30506     + #1#2
30507     \fi:
30508     \exp_after:wN \__fp_sep:
30509 }
30510 \cs_new:Npn \__fp_randint_split_aux:w #1 \__fp_sep:
30511 {
30512     #1 \exp_after:wN \__fp_sep:
30513     \int_value:w \__fp_int_eval:w - #1 * 16384
30514 }
30515 \cs_new:Npn \__fp_randint_wide_aux:w
30516 #1 \__fp_sep: #2 \__fp_sep: #3 \__fp_sep: #4 \__fp_sep:

```

```

30517     #5\__fp_sep:#6\__fp_sep:#7\__fp_sep: .
30518 {
30519     \exp_after:wN \__fp_randint_wide_auxii:w
30520     \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +
30521     (#5 * #4 + #6 * #3 + #7 * #1 +
30522     (#5 * #2 + #7 * #3 +
30523     (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
30524     ) / 16384 \exp_after:wN \__fp_sep:
30525     \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 \__fp_sep:
30526     #1 \__fp_sep: #5 \__fp_sep:
30527 }
30528 \cs_new:Npn \__fp_randint_wide_auxii:w
30529 #1\__fp_sep: #2\__fp_sep: #3\__fp_sep: #4\__fp_sep:
30530 {
30531     \if_int_odd:w 0
30532     \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
30533     \if_int_compare:w #4 = \c_zero_int 1 \fi:
30534     \if_int_compare:w #3 = 16383 ~ 1 \fi:
30535     \exp_stop_f:
30536     \exp_after:wN \prg_break:
30537     \fi:
30538     \if_int_compare:w #4 < 8 \exp_stop_f:
30539     + #4 * #3 * 16384
30540     \else:
30541     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
30542     \fi:
30543     + #1
30544     \prg_break_point:
30545 }

```

(End of definition for __kernel_randint:nn and others.)

\int_rand:n Similar to \int_rand:nn, but needs fewer checks.

__fp_randint:n

```

30546 \cs_new:Npn \int_rand:n #1
30547 {
30548     \int_eval:n
30549     { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
30550 }
30551 \cs_new:Npn \__fp_randint:n #1
30552 {
30553     \if_int_compare:w #1 < \c_one_int
30554     \msg_expandable_error:nnnn
30555     { kernel } { randint-backward-range } { 1 } {#1}
30556     \__fp_randint:ww #1\__fp_sep: 1\__fp_sep:
30557     \else:
30558     \if_int_compare:w #1 > \c__kernel_randint_max_int
30559     \__kernel_randint:nn { 1 } {#1}
30560     \else:
30561     \__kernel_randint:n {#1}
30562     \fi:
30563     \fi:
30564 }

```

(End of definition for \int_rand:n and __fp_randint:n. This function is documented on page 183.)

30565 `</code>`

Chapter 85

l3fp-types implementation

```
30566 <*code>
30567 <@@=fp>
```

85.1 Support for types

Despite lack of documentation, the l3fp internals support types. Each additional type must define

- `\s__fp_<type>` and `\__fp_<type>_chk:w`;
- `\__fp_exp_after_<type>_f:nw`;
- `\__fp_<type>_to_<out>:w` for `<out>` among `decimal`, `scientific`, `tl`;

and may define

- `\__fp_<type>_to_int:w` and `\__fp_<type>_to_dim:w`;
- `\__fp_<op>_<type>_o:w` for any of the `<op>` that the type implements, among `acos`, `acsc`, `asec`, `asin`, `cos`, `cot`, `csc`, `exp`, `ln`, `not`, `sec`, `set_sign`, `sin`, `tan`;
- `\__fp_<type1>_<op>_<type2>_o:ww` for `<op>` among `^*/-+&|` and for every pair of types;
- `\__fp_<type1>_bcmp_<type2>:ww` for every pair of types.

The latter is set up in l3fp-logic.

85.2 Dispatch according to the type

```
\__fp_types_cs_to_op:N From \__fp_<op>_o:w produce <op>, otherwise ?.
\__fp_types_cs_to_op_auxi:wwwn
30568 \cs_new:Npe \__fp_types_cs_to_op:N #1
30569 {
30570   \exp_not:N \exp_after:wN \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
30571   \exp_not:N \token_to_str:N #1 \s__fp_mark
30572   \exp_not:N \__fp_use_i_delimit_by_s_stop:nw
30573   \tl_to_str:n { __fp_o:w } \s__fp_mark
30574   { \exp_not:N \__fp_use_i_delimit_by_s_stop:nw ? }
```

```

30575         \s__fp_stop
30576     }
30577 \use:e
30578 {
30579     \cs_new:Npn \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
30580         #1 \tl_to_str:n { __fp_ } #2
30581         \tl_to_str:n { _o:w } #3 \s__fp_mark #4 { #4 {#2} }
30582 }

```

(End of definition for __fp_types_cs_to_op:N and __fp_types_cs_to_op_auxi:wwwn.)

```

\__fp_types_unary:NNw         \__fp_types_unary:NNw \__fp_⟨function⟩_o:w
\__fp_types_unary_auxi:nNw    ⟨token⟩ ⟨operand⟩ @
\__fp_types_unary_auxii:NnNw
30583 \cs_new:Npn \__fp_types_unary:NNw #1
30584 {
30585     \exp_args:Nf \__fp_types_unary_auxi:nNw
30586     { \__fp_types_cs_to_op:N #1 }
30587 }
30588 \cs_new:Npn \__fp_types_unary_auxi:nNw #1#2#3
30589 {
30590     \exp_after:wN \__fp_types_unary_auxii:NnNw
30591     \cs:w __fp_#1 \__fp_type_from_scan:N #3 _o:w \cs_end:
30592     {#1}
30593     #2#3
30594 }
30595 \cs_new:Npn \__fp_types_unary_auxii:NnNw #1#2#3
30596 {
30597     \token_if_eq_meaning:NNTF \scan_stop: #1
30598     { \__fp_invalid_operation_o:nw {#2} }
30599     { #1 #3 }
30600 }

```

(End of definition for __fp_types_unary:NNw, __fp_types_unary_auxi:nNw, and __fp_types_unary_auxii:NnNw.)

```

\__fp_types_binary:Nww        \__fp_types_binary:Nww \__fp_⟨binop⟩_o:ww
\__fp_types_binary_auxi:Nww    ⟨operand1⟩ ⟨operand2⟩ @
\__fp_types_binary_auxii:NNww
30601 \cs_new:Npn \__fp_types_binary:Nww #1
30602 {
30603     \exp_last_unbraced:Nf \__fp_types_binary_auxi:Nww
30604     { \__fp_types_cs_to_op:N #1 }
30605 }
30606 \cs_new:Npn \__fp_types_binary_auxi:Nww #1#2#3\__fp_sep: #4#5\__fp_sep: @
30607 {
30608     \exp_after:wN \__fp_types_binary_auxii:NNww
30609     \cs:w
30610     __fp
30611     \__fp_type_from_scan:N #2
30612     _#1
30613     \__fp_type_from_scan:N #4
30614     _o:ww
30615     \cs_end:
30616     #1 #2#3\__fp_sep: #4#5\__fp_sep:
30617 }

```

```

30618 \cs_new:Npn \__fp_types_binary_auxii:NNww #1#2
30619 {
30620   \token_if_eq_meaning:NNTF \scan_stop: #1
30621   { \__fp_invalid_operation_o:Nww #2 }
30622   {#1}
30623 }

```

(End of definition for __fp_types_binary:Nww, __fp_types_binary_auxi:Nww, and __fp_types_binary_auxii:NNww.)

```

30624 \</code>

```


Chapter 86

l3fp-symbolic implementation

30625 $\langle *code \rangle$

30626 $\langle @@=fp \rangle$

86.1 Misc

`\l__fp_symbolic_fp` Scratch floating point.

30627 `\fp_new:N \l__fp_symbolic_fp`

(End of definition for \l__fp_symbolic_fp.)

86.2 Building blocks for expressions

Every symbolic expression has the form `\s__fp_symbolic \__fp_symbolic_chk:w  $\langle operation \rangle$  , { $\langle operands \rangle$ }  $\langle junk \rangle$  \__fp_sep:` where the $\langle operation \rangle$ is a list of N-type tokens, the $\langle operands \rangle$ is an array of floating point objects, and the $\langle junk \rangle$ (typically empty) is to be discarded. If the outermost operator (last to be evaluated) is unary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_unary:NNw \__fp_<op>_o:w <token> ,
{ <operand> } <junk> \__fp_sep:
```

where the $\langle op \rangle$ is a unary operation (`set_sign`, `cos`, ...), and the $\langle token \rangle$ and $\langle operand \rangle$ are used as arguments for `\__fp_<op>_o:w` (or the type-specific analog of this function). If the outermost operator is binary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_binary:Nww \__fp_<op>_o:ww ,
{ <operand1> <operand2> } <junk> \__fp_sep:
```

where the $\langle op \rangle$ is an operation (`+`, `&`, ...), and `\__fp_<op>_o:ww` receives the $\langle operands \rangle$ as arguments. If the expression is a user-defined function applied to some arguments it is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_function_o:w \__fp_<identifier>_o:w ,
{ <operands> } <junk> \__fp_sep:
```

If the expression consists of a single variable, it is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_variable_o:w <identifier> ,
{ } <junk> \__fp_sep:
```

Symbolic expressions are stored in a prefix form. When encountering a symbolic expression in a floating point computation, we attempt to evaluate the operands as much as possible, and if that yields floating point numbers rather than expressions, we apply the operator which follows (if the function is known).

For instance, the expression $a + b * \sin(c)$ is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_binary:Nww \__fp+_o:ww ,
{
  \s__fp_symbolic \__fp_symbolic_chk:w
  \__fp_variable_o:w a , { } \__fp_sep:
  \s__fp_symbolic \__fp_symbolic_chk:w
  \__fp_types_binary:Nww \__fp*_o:ww ,
  {
    \s__fp_symbolic \__fp_symbolic_chk:w
    \__fp_variable_o:w b , { } \__fp_sep:
    \s__fp_symbolic \__fp_symbolic_chk:w
    \__fp_types_unary:NNw \__fp_sin_o:w \use_i:nn ,
    {
      \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_variable_o:w c , { } \__fp_sep:
    } \__fp_sep:
  } \__fp_sep:
} \__fp_sep:
```

\s__fp_symbolic Scan mark indicating the start of a symbolic expression.

```
30628 \scan_new:N \s__fp_symbolic
```

(End of definition for \s__fp_symbolic.)

__fp_symbolic_chk:w Analog of __fp_chk:w for symbolic expressions.

```
30629 \cs_new_protected:Npn \__fp_symbolic_chk:w #1,#2#3\__fp_sep:
30630 {
30631   \msg_error:nne { fp } { misused-fp }
30632   {
30633     \__fp_to_tl_dispatch:w
30634     \s__fp_symbolic \__fp_symbolic_chk:w #1,{#2}\__fp_sep:
30635   }
30636 }
```

(End of definition for __fp_symbolic_chk:w.)

86.3 Expanding after a symbolic expression

`\_fp_if_has_symbolic:nTF` Tests if #1 contains `\s_fp_symbolic` at top-level. This test should be precise enough to determine if a given array contains a symbolic expression or only consists of floating points. Used in `\_fp_exp_after_symbolic_f:nw`.

```

30637 \cs_new:Npn \_fp_if_has_symbolic:nTF #1
30638 {
30639     \_fp_if_has_symbolic_aux:w
30640     #1 \s_fp_mark \use_i:nn
30641     \s_fp_symbolic \s_fp_mark \use_ii:nn
30642     \s_fp_stop
30643 }
30644 \cs_new:Npn \_fp_if_has_symbolic_aux:w
30645     #1 \s_fp_symbolic #2 \s_fp_mark #3#4 \s_fp_stop { #3 }

```

(End of definition for `\_fp_if_has_symbolic:nTF` and `\_fp_if_has_symbolic_aux:w`.)

`\_fp_exp_after_symbolic_f:nw` This function does two things: trigger an f-expansion of the argument #1 after the following symbolic expression, and evaluate all pieces of the expression which can be evaluated.

```

30646 \cs_new:Npn \_fp_exp_after_symbolic_f:nw
30647     #1 \s_fp_symbolic \_fp_symbolic_chk:w #2, #3#4\_fp_sep:
30648 {
30649     \exp_after:wN \_fp_exp_after_symbolic_aux:w
30650     \exp:w
30651     \_fp_exp_after_symbolic_loop:N #2
30652     { , \exp:w \use_none:nn }
30653     \exp_after:wN \exp_end: \exp_after:wN
30654     {
30655         \exp:w \exp_end_continue_f:w
30656         \_fp_exp_after_array_f:w #3 \s_fp_expr_stop
30657         \exp_after:wN
30658     }
30659     \exp_after:wN \_fp_sep:
30660     \exp:w \exp_end_continue_f:w #1
30661 }
30662 \cs_new:Npn \_fp_exp_after_symbolic_aux:w #1, #2\_fp_sep:
30663 {
30664     \_fp_if_has_symbolic:nTF {#2}
30665     { \s_fp_symbolic \_fp_symbolic_chk:w #1, {#2} \_fp_sep: }
30666     { #1 #2 @ \prg_do_nothing: }
30667 }
30668 \cs_new:Npn \_fp_exp_after_symbolic_loop:N #1
30669 {
30670     \exp_after:wN \exp_end:
30671     \exp_after:wN #1
30672     \exp:w
30673     \_fp_exp_after_symbolic_loop:N
30674 }

```

(End of definition for `\_fp_exp_after_symbolic_f:nw`, `\_fp_exp_after_symbolic_aux:w`, and `\_fp_exp_after_symbolic_loop:N`.)

86.4 Applying infix operators to expressions

Used when applying infix operators to expressions.

```

\__fp_symbolic_binary_o:Nww
30675 \cs_new:Npn \__fp_symbolic_binary_o:Nww #1 #2\__fp_sep: #3\__fp_sep:
30676 {
30677   \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
30678   \s__fp_symbolic \__fp_symbolic_chk:w
30679   \__fp_types_binary:Nww #1 , { #2\__fp_sep: #3\__fp_sep: } \__fp_sep:
30680 }

```

(End of definition for __fp_symbolic_binary_o:Nww.)
 $\sim$ A Hack! $\sim$ A Hack! $\sim$ A Hack!

```

\__fp_symbolic+_symbolic_o:ww
\__fp_symbolic+_o:ww
\__fp+_symbolic_o:ww
\__fp_symbolic-_symbolic_o:ww
\__fp_symbolic-_o:ww
\__fp-_symbolic_o:ww
\__fp_symbolic*_symbolic_o:ww
\__fp_symbolic*_o:ww
\__fp*_symbolic_o:ww
\__fp_symbolic/_symbolic_o:ww
\__fp_symbolic/_o:ww
\__fp/_symbolic_o:ww
\__fp_symbolic^symbolic_o:ww
\__fp_symbolic^_o:ww
\__fp^symbolic_o:ww
\__fp_symbolic|symbolic_o:ww
\__fp_symbolic|_o:ww
\__fp|symbolic_o:ww
\__fp_symbolic&symbolic_o:ww
\__fp\symbolic&symbolic_o:ww
\__fp&symbolic_o:ww
30681 \cs_set_protected:Npn \__fp_tmp:w #1#2
30682 {
30683   \cs_new:cpn
30684     { __fp_symbolic_#2_symbolic_o:ww }
30685     { \__fp_symbolic_binary_o:Nww #1 }
30686   \cs_new_eq:cc
30687     { __fp_symbolic_#2_o:ww }
30688     { __fp_symbolic_#2_symbolic_o:ww }
30689   \cs_new_eq:cc
30690     { __fp_#2_symbolic_o:ww }
30691     { __fp_symbolic_#2_symbolic_o:ww }
30692 }
30693 \tl_map_inline:nn { + - * / ^ & | } % |
30694 { \exp_args:Nc \__fp_tmp:w { __fp_#1_o:ww } {#1} }

```

(End of definition for __fp_symbolic+_symbolic_o:ww and others.)

86.5 Applying prefix functions to expressions

Used when applying infix operators to expressions.

```

\__fp_symbolic_unary_o:NNw
30695 \cs_new:Npn \__fp_symbolic_unary_o:NNw #1#2#3\__fp_sep: @
30696 {
30697   \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
30698   \s__fp_symbolic \__fp_symbolic_chk:w
30699   \__fp_types_unary:NNw #1#2 , { #3\__fp_sep: } \__fp_sep:
30700 }

```

(End of definition for __fp_symbolic_unary_o:NNw.)

```

\__fp_symbolic_acos_o:w
\__fp_symbolic_acsc_o:w
\__fp_symbolic_asec_o:w
\__fp_symbolic_asin_o:w
\__fp_symbolic_cos_o:w
\__fp_symbolic_cot_o:w
\__fp_symbolic_csc_o:w
\__fp_symbolic_exp_o:w
\__fp_symbolic_fact_o:w
\__fp_symbolic_ln_o:w
\__fp_symbolic_not_o:w
\__fp_symbolic_sec_o:w
\__fp_symbolic_set_sign_o:w
\__fp_symbolic_sign_o:w
\__fp_symbolic_sin_o:w
\__fp_symbolic_tan_o:w
30701 \tl_map_inline:nn
30702 {
30703   {acos} {acsc} {asec} {asin} {cos} {cot} {csc} {exp} {fact} {ln}
30704   {not} {sec} {set_sign} {sin} {sign} {sqrt} {tan}
30705 }
30706 {
30707   \cs_new:cpe { __fp_symbolic_#1_o:w }
30708   {
30709     \exp_not:N \__fp_symbolic_unary_o:NNw
30710     \exp_not:c { __fp_#1_o:w }

```

```

30711     }
30712 }

```

(End of definition for `\_fp_symbolic_acos_o:w` and others.)

86.6 Conversions

Symbolic expressions cannot be converted to decimal, integer, or scientific notation unless they can be evaluated all the way.

```

\_fp_symbolic_to_decimal:w
\_fp_symbolic_to_int:w
\_fp_symbolic_to_scientific:w
\_fp_symbolic_convert:wnnN
30713 \cs_set_protected:Npn \_fp_tmp:w #1#2#3
30714 {
30715   \cs_new:cpn { \_fp_symbolic_to_#1:w }
30716   {
30717     \exp_after:wN \_fp_symbolic_convert:wnnN
30718     \exp:w \exp_end_continue_f:w
30719     \_fp_exp_after_symbolic_f:nw { { #2 } { fp_to_#1 } #3 }
30720   }
30721 }
30722 \_fp_tmp:w { decimal } { 0 } \_fp_to_decimal_dispatch:w
30723 \_fp_tmp:w { int } { 0 } \_fp_to_int_dispatch:w
30724 \_fp_tmp:w { scientific } { nan } \_fp_to_scientific_dispatch:w
30725 \cs_new:Npn \_fp_symbolic_convert:wnnN #1#2\_fp_sep: #3#4#5
30726 {
30727   \str_if_eq:nnTF {#1} { \s\_fp_symbolic }
30728   { \_fp_invalid_operation:nnw {#3} {#4} #1#2\_fp_sep: }
30729   { #5 #1#2\_fp_sep: }
30730 }

```

(End of definition for `\_fp_symbolic_to_decimal:w` and others.)

```

\_fp_symbolic_cs_arg_to_fn:NN
\_fp_symbolic_op_arg_to_fn:nN
30731 \cs_new:Npn \_fp_symbolic_cs_arg_to_fn:NN #1
30732 {
30733   \exp_args:Nf \_fp_symbolic_op_arg_to_fn:nN
30734   { \_fp_types_cs_to_op:N #1 }
30735 }
30736 \cs_new:Npn \_fp_symbolic_op_arg_to_fn:nN #1#2
30737 {
30738   \str_case:nnF { #1 #2 }
30739   {
30740     { not ? } { ! }
30741     { set_sign 0 } { abs }
30742     { set_sign 2 } { - }
30743   }
30744   {
30745     \token_if_eq_meaning:NNTF #2 \use_ii:nn
30746     { #1 d } {#1}
30747   }
30748 }

```

(End of definition for `\_fp_symbolic_cs_arg_to_fn:NN` and `\_fp_symbolic_op_arg_to_fn:nN`.)

```

__fp_symbolic_to_tl:w      Converting a symbolic expression to a token list is possible.
  __fp_symbolic_unary_to_tl:NNw
  __fp_symbolic_binary_to_tl:Nww
  __fp_symbolic_function_to_tl:Nw
30749 \cs_new:Npn \__fp_symbolic_to_tl:w
30750   \s__fp_symbolic \__fp_symbolic_chk:w #1#2, #3#4\__fp_sep:
30751   {
30752     \str_case:nnTF {#1}
30753     {
30754       { \__fp_types_unary:NNw } { \__fp_symbolic_unary_to_tl:NNw }
30755       { \__fp_types_binary:Nww } { \__fp_symbolic_binary_to_tl:Nww }
30756       { \__fp_function_o:w } { \__fp_symbolic_function_to_tl:Nw }
30757     }
30758     { #2, #3 @ }
30759     { \tl_to_str:n {#2} }
30760   }
30761 \cs_new:Npn \__fp_symbolic_unary_to_tl:NNw #1#2 , #3 @
30762   {
30763     \use:e
30764     {
30765       \__fp_symbolic_cs_arg_to_fn:NN #1#2
30766       ( \__fp_to_tl_dispatch:w #3 )
30767     }
30768   }
30769 \cs_new:Npn \__fp_symbolic_binary_to_tl:Nww #1, #2\__fp_sep: #3\__fp_sep: @
30770   {
30771     \use:e
30772     {
30773       ( \__fp_to_tl_dispatch:w #2\__fp_sep: )
30774       \__fp_types_cs_to_op:N #1
30775       ( \__fp_to_tl_dispatch:w #3\__fp_sep: )
30776     }
30777   }
30778 \cs_new:Npn \__fp_symbolic_function_to_tl:Nw #1, #2@
30779   {
30780     \use:e
30781     {
30782       \__fp_types_cs_to_op:N #1
30783       ( \__fp_array_to_clist:n {#2} )
30784     }
30785   }

```

(End of definition for __fp_symbolic_to_tl:w and others.)

86.7 Identifiers

Functions defined here are not necessarily tied to symbolic expressions.

__fp_id_if_invalid:nTF If #1 contains a space, it is not a valid identifier. Otherwise, loop through letters in #1: if it is not a letter, break the loop and return true. If the end of the loop is reached without finding any non-letter, return false. Note #1 must be a str (i.e., resulted from \tl_to_str:n).

```

30786 \prg_new_protected_conditional:Npnn
30787   \__fp_id_if_invalid:n #1 { T , F , TF }
30788   {
30789     \tl_if_empty:nTF {#1}

```

```

30790     { \prg_return_true: }
30791     {
30792         \tl_if_in:nnTF { #1 } { ~ }
30793         { \prg_return_true: }
30794         {
30795             \__fp_id_if_invalid_aux:N #1
30796             { ? \prg_break:n \prg_return_false: }
30797             \prg_break_point:
30798         }
30799     }
30800 }
30801 \cs_new:Npn \__fp_id_if_invalid_aux:N #1
30802 {
30803     \use_none:n #1
30804     \int_compare:nF { 'a <= '#1 <= 'z }
30805     {
30806         \int_compare:nF { 'A <= '#1 <= 'Z }
30807         { \prg_break:n \prg_return_true: }
30808     }
30809     \__fp_id_if_invalid_aux:N
30810 }

```

(End of definition for __fp_id_if_invalid:nTF and __fp_id_if_invalid_aux:N.)

86.8 Declaring variables and assigning values

__fp_variable_o:w We do not use \exp_last_unbraced:Nv to extract the value of \l__fp_variable_#1_fp because in \fp_set_variable:nn we define this fp variable to be something which f-expands to an actual floating point, rather than a genuine floating point.

```

30811 \cs_new:Npn \__fp_variable_o:w #1 @ #2
30812 {
30813     \fp_if_exist:cTF { l__fp_variable_#1_fp }
30814     {
30815         \exp_last_unbraced:Nf \__fp_exp_after_array_f:w
30816         { \use:c { l__fp_variable_#1_fp } } \s__fp_expr_stop
30817         \exp_after:wN \exp_stop_f: #2
30818     }
30819     {
30820         \token_if_eq_meaning:NNTF #2 \prg_do_nothing:
30821         {
30822             \s__fp_symbolic \__fp_symbolic_chk:w
30823             \__fp_variable_o:w #1 , { } \__fp_sep:
30824         }
30825         {
30826             \exp_after:wN \s__fp_symbolic
30827             \exp_after:wN \__fp_symbolic_chk:w
30828             \exp_after:wN \__fp_variable_o:w
30829             \exp:w
30830             \__fp_exp_after_symbolic_loop:N #1
30831             { , \exp:w \use_none:nn }
30832             \exp_after:wN \exp_end:
30833             \exp_after:wN { \exp_after:wN } \exp_after:wN \__fp_sep:
30834             #2

```

```

30835     }
30836   }
30837 }

```

(End of definition for _fp_variable_o:w.)

```

\_fp\_variable\_set\_parsing:Nn
\_fp\_variable\_set\_parsing\_aux:NNn

```

```

30838 \cs_new_protected:Npn \_fp\_variable\_set\_parsing:Nn #1#2
30839 {
30840   \cs_set:Npn \_fp\_tmp:w
30841   {
30842     \_fp\_exp\_after\_symbolic\_f:nw { \_fp\_parse\_infix:NN }
30843     \s\_fp\_symbolic \_fp\_symbolic\_chk:w
30844     \_fp\_variable\_o:w #2 , { } \_fp\_sep:
30845   }
30846   \exp\_args:NNc \_fp\_variable\_set\_parsing\_aux:NNn #1
30847   { \_fp\_parse\_word\_#2:N } {#2}
30848 }
30849 \cs_new_protected:Npn \_fp\_variable\_set\_parsing\_aux:NNn #1#2#3
30850 {
30851   \cs_if_eq:N NF #2 \_fp\_tmp:w
30852   {
30853     \cs_if_exist:N TF #2
30854     {
30855       \msg_warning:nnnn
30856       { fp } { id-used-elsewhere } {#3} { variable }
30857       #1 #2 \_fp\_tmp:w
30858     }
30859     {
30860       \cs_new_eq:NN #2 \scan_stop: % to declare the function
30861       #1 #2 \_fp\_tmp:w
30862     }
30863   }
30864 }

```

(End of definition for _fp_variable_set_parsing:Nn and _fp_variable_set_parsing_aux:NNn.)

```

\_fp\_clear\_variable:n We need local undefining, so have to do it low-level. \_fp\_clear\_variable\_aux:n is
\_fp\_clear\_variable:n needed by \_fp\_set\_function:Nnnn to skip \_fp\_id\_if\_invalid:nTF.
\_fp\_clear\_variable\_aux:n

```

```

30865 \cs_new_protected:Npn \fp\_clear\_variable:n #1
30866 {
30867   \exp\_args:No \_fp\_clear\_variable:n { \tl\_to\_str:n {#1} }
30868 }
30869 \cs_new_protected:Npn \_fp\_clear\_variable:n #1
30870 {
30871   \_fp\_id\_if\_invalid:nTF {#1}
30872   { \msg_error:nnn { fp } { id-invalid } {#1} }
30873   { \_fp\_clear\_variable\_aux:n {#1} }
30874 }
30875 \cs_new_protected:Npn \_fp\_clear\_variable\_aux:n #1
30876 {
30877   \cs_set_eq:cN { l\_fp\_variable\_#1\_fp } \tex\_undefined:D
30878   \_fp\_variable\_set\_parsing:Nn \cs\_set\_eq:NN {#1}
30879 }

```


(End of definition for `\fp_clear_variable:n`, `\_fp_clear_variable:n`, and `\__fp_clear_variable_aux:n`. This function is documented on page 275.)

`\fp_new_variable:n` Check that #1 is a valid identifier. If the identifier is already in use, complain. Then set `\__fp_new_variable:n` `\__fp_parse_word_#1:N` to use `\_fp_variable_o:w`.

```

30880 \cs_new_protected:Npn \fp_new_variable:n #1
30881 {
30882   \exp_args:No \__fp_new_variable:n { \tl_to_str:n {#1} }
30883 }
30884 \cs_new_protected:Npn \__fp_new_variable:n #1
30885 {
30886   \__fp_id_if_invalid:nTF {#1}
30887   { \msg_error:nnn { fp } { id-invalid } {#1} }
30888   {
30889     \cs_if_exist:cT { __fp_parse_word_#1:N }
30890     {
30891       \msg_error:nnn
30892       { fp } { id-already-defined } {#1}
30893       \cs_undefine:c { __fp_parse_word_#1:N }
30894       \cs_set_eq:cN { l__fp_variable_#1_fp } \tex_undefined:D
30895     }
30896     \__fp_variable_set_parsing:Nn \cs_gset_eq:NN {#1}
30897   }
30898 }

```

(End of definition for `\fp_new_variable:n` and `\__fp_new_variable:n`. This function is documented on page 274.)

`\l__fp_symbolic_flag` Refuse invalid identifiers. If the variable does not exist yet, raise an error and define it just as in `\fp_set_variable:nn` (but without unnecessary checks). Then evaluate #2. If `\fp_set_variable:nn` the result contains the identifier #1, we would later get a loop in cases such as

```

\fp_set_variable:nn {A} {A}
\fp_show:n {A}

```

To detect this, define `\l__fp_variable_#1_fp` to raise an internal flag and evaluate to `nan`. Then re-evaluate `\l__fp_symbolic_flag`, and store the result in #1. If the flag is raised, #1 was present in `\l__fp_symbolic_flag`. In all cases, the #1-free result ends up in `\l__fp_variable_#1_fp`.

```

30899 \flag_new:N \l__fp_symbolic_flag
30900 \cs_new_protected:Npn \fp_set_variable:nn #1
30901 {
30902   \exp_args:No \__fp_set_variable:nn { \tl_to_str:n {#1} }
30903 }
30904 \cs_new_protected:Npn \__fp_set_variable:nn #1#2
30905 {
30906   \__fp_id_if_invalid:nTF {#1}
30907   { \msg_error:nnn { fp } { id-invalid } {#1} }
30908   {
30909     \cs_if_exist:cF { __fp_parse_word_#1:N }
30910     {
30911       \msg_error:nnn {fp} { id-undefined } {#1}
30912       \__fp_variable_set_parsing:Nn \cs_set_eq:NN {#1}
30913     }

```

```

30914 \fp_set:Nn \l__fp_symbolic_fp {#2}
30915 \cs_set_nopar:cpn { l__fp_variable_#1_fp }
30916 { \flag_ensure_raised:N \l__fp_symbolic_flag \c_nan_fp }
30917 \flag_clear:N \l__fp_symbolic_flag
30918 \fp_set:cn { l__fp_variable_#1_fp } { \l__fp_symbolic_fp }
30919 \flag_if_raised:NT \l__fp_symbolic_flag
30920 {
30921 \msg_error:nneee { fp } { id-loop }
30922 { #1 }
30923 { \tl_to_str:n {#2} }
30924 { \fp_to_tl:N \l__fp_symbolic_fp }
30925 }
30926 }
30927 }

```

(End of definition for `\l__fp_symbolic_flag`, `\fp_set_variable:nn`, and `\__fp_set_variable:nn`. This variable is documented on page 274.)

86.9 Messages

```

30928 \msg_new:nnnn { fp } { id-invalid }
30929 { Floating-point~identifier~'~#1'~invalid. }
30930 {
30931 LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'~#1'~
30932 but~this~may~only~contain~ASCII~letters.
30933 }
30934 \msg_new:nnnn { fp } { id-already-defined }
30935 { Floating-point~identifier~'~#1'~already-defined. }
30936 {
30937 LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'~#1'~
30938 but~this~name~has~already~been~used~elsewhere.
30939 }
30940 \msg_new:nnnn { fp } { id-undefined }
30941 { Floating-point~identifier~'~#1'~is~undefined. }
30942 {
30943 LaTeX~has~been~asked~to~set~a~floating~point~identifier~'~#1'~
30944 but~this~name~has~not~been~declared.
30945 }
30946 \msg_new:nnnn { fp } { id-used-elsewhere }
30947 { Floating-point~identifier~'~#1'~already-used~for~something~else. }
30948 {
30949 LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'~#1'~
30950 but~this~name~is~used,~and~is~not~a~user-defined~#2.
30951 }
30952 \msg_new:nnnn { fp } { id-loop }
30953 { Variable~'~#1'~used~in~the~definition~of~'~#1'~. }
30954 {
30955 LaTeX~has~been~asked~to~set~the~floating~point~identifier~'~#1'~
30956 to~the~expression~'~#2'~.~Evaluating~this~expression~yields~'~#3'~,~
30957 which~contains~'~#1'~itself.
30958 }

```

86.10 Road-map

The following functions are not implemented: `min`, `max`, `?:`, comparisons, `round`, `atan`, `acot`.

30959 `</code>`

Chapter 87

l3fp-functions implementation

```
30960 <*code>
30961 <@@=fp>
```

87.1 Declaring functions

```
\fp_new_function:n
__fp_new_function:n
30962 \cs_new_protected:Npn \fp_new_function:n #1
30963   { \exp_args:No \__fp_new_function:n { \tl_to_str:n {#1} } }
30964 \cs_new_protected:Npn \__fp_new_function:n #1
30965   {
30966     \__fp_id_if_invalid:nTF {#1}
30967     { \msg_error:nnn { fp } { id-invalid } {#1} }
30968     {
30969       \cs_if_exist:cT { __fp_parse_word_#1:N }
30970       {
30971         \msg_error:nnn
30972         { fp } { id-already-defined } {#1}
30973         \cs_undefine:c { __fp_parse_word_#1:N }
30974         \cs_undefine:c { __fp_#1_o:w }
30975       }
30976       \__fp_function_set_parsing:Nn \cs_gset_eq:NN {#1}
30977     }
30978   }
```

(End of definition for `\fp_new_function:n` and `\__fp_new_function:n`. This function is documented on page 275.)

```
\__fp_function_set_parsing:Nn
__fp_function_set_parsing_aux:NNn
30979 \cs_new_protected:Npn \__fp_function_set_parsing:Nn #1#2
30980   {
30981     \exp_args:NNc \__fp_function_set_parsing_aux:NNn #1
30982     { __fp_parse_word_#2:N } {#2}
30983   }
30984 \cs_new_protected:Npn \__fp_function_set_parsing_aux:NNn #1#2#3
30985   {
30986     \cs_set:Npe \__fp_tmp:w
30987     {
30988       \exp_not:N \__fp_parse_function:NNN
```

```

30989         \exp_not:N \__fp_function_o:w
30990         \exp_not:c { __fp_#3_o:w }
30991     }
30992     \cs_if_eq:NNF #2 \__fp_tmp:w
30993     {
30994         \cs_if_exist:NTF #2
30995         {
30996             \msg_warning:nnnn
30997             { fp } { id-used-elsewhere } {#3} { function }
30998             #1 #2 \__fp_tmp:w
30999         }
31000         {
31001             \cs_new_eq:NN #2 \scan_stop: % to declare the function
31002             #1 #2 \__fp_tmp:w
31003         }
31004     }
31005 }

```

(End of definition for __fp_function_set_parsing:Nn and __fp_function_set_parsing_aux:NNn.)

__fp_function_o:w

```

31006 \cs_new:Npn \__fp_function_o:w #1#2 @
31007 {
31008     \cs_if_exist:NTF #1
31009     { #1 #2 @ }
31010     {
31011         \exp_after:wN \s__fp_symbolic
31012         \exp_after:wN \__fp_symbolic_chk:w
31013         \exp_after:wN \__fp_function_o:w
31014         \exp_after:wN #1
31015         \exp_after:wN ,
31016         \exp_after:wN {
31017             \exp:w \exp_end_continue_f:w
31018             \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
31019             \exp_after:wN
31020         }
31021         \exp_after:wN \__fp_sep:
31022     }
31023 }

```

(End of definition for __fp_function_o:w.)

87.2 Defining functions by their expression

\l__fp_function_arg_int Labels the arguments of a function being defined.

```

31024 \int_new:N \l__fp_function_arg_int

```

(End of definition for \l__fp_function_arg_int.)

\fp_set_function:nnn \fp_set_function:nnn {<identifier>}
 __fp_set_function:Nnnn {<comma-list of variables>} {<expression>}

Sets the <identifier> to stand for a function which expects some arguments defined by the <comma-list of variables>, and evaluates to the <expression>.

```

31025 \cs_new_protected:Npn \fp_set_function:nnn #1
31026 {
31027   \exp_args:NNo \__fp_set_function:Nnnn \cs_set_eq:cN
31028   { \tl_to_str:n {#1} }
31029 }
31030 \cs_new_protected:Npn \__fp_set_function:Nnnn #1#2#3#4
31031 {
31032   \__fp_id_if_invalid:nTF {#2}
31033   { \msg_error:nnn { fp } { id-invalid } {#2} }
31034   {
31035     \cs_if_exist:cF { __fp_parse_word_#2:N }
31036     {
31037       \msg_error:nnn {fp} { id-undefined } {#2}
31038       \__fp_function_set_parsing:Nn \cs_set_eq:NN {#2}
31039     }
31040     \group_begin:
31041     \int_zero:N \l__fp_function_arg_int
31042     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#3} }
31043     {
31044       \int_incr:N \l__fp_function_arg_int
31045       \exp_args:Ne \__fp_clear_variable_aux:n
31046       {
31047         \c_underscore_str \tex_romannumeral:D \l__fp_function_arg_int
31048       }
31049       \fp_clear_variable:n {##1}
31050       \cs_set_nopar:cpe { l__fp_variable_##1_fp }
31051       {
31052         \exp_not:N \s__fp_symbolic
31053         \exp_not:N \__fp_symbolic_chk:w
31054         \exp_not:N \__fp_function_arg_o:w
31055         \int_use:N \l__fp_function_arg_int
31056         #####1 , { } \__fp_sep:
31057       }
31058     }
31059     \cs_set:Npn \__fp_function_arg_o:w ##1 @
31060     {
31061       \exp_after:wN \s__fp_symbolic
31062       \exp_after:wN \__fp_symbolic_chk:w
31063       \exp_after:wN \__fp_function_arg_o:w
31064       \tex_romannumeral:D
31065       \__fp_exp_after_symbolic_loop:N ##1
31066       { , \tex_romannumeral:D \use_none:nn }
31067       \exp_after:wN \c_zero_int
31068       \exp_after:wN { \exp_after:wN } \exp_after:wN \__fp_sep:
31069     }
31070     \fp_set:Nn \l__fp_symbolic_fp {#4}
31071     \use:e
31072     {
31073       \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 }
31074       { \exp_not:o { \l__fp_symbolic_fp } }
31075     }
31076     \use:e
31077     {
31078       \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 @ }

```

```

31079         {
31080             \exp_not:N \__fp_exp_after_symbolic_f:nw
31081             \exp_not:n { { \exp_after:wN \exp_stop_f: } }
31082             \exp_not:o { \__fp_tmp:w { . , {##1} } }
31083         }
31084     }
31085     \group_end:
31086     #1 { __fp_#2_o:w } \__fp_tmp:w
31087 }
31088 }

```

```

\__fp_function_arg_o:w 31089 \cs_new:Npn \__fp_function_arg_o:w #1. #2
\__fp_function_arg_few:w 31090 {
\__fp_function_arg_get:w 31091     \if_meaning:w @ #2
31092         \exp_after:wN \__fp_function_arg_few:w
31093     \fi:
31094     \if_int_compare:w #1 = \c_one_int
31095         \exp_after:wN \__fp_function_arg_get:w
31096     \fi:
31097     \__fp_use_i_until_s:nw
31098     {
31099         \exp_after:wN \__fp_function_arg_o:w
31100         \int_value:w \int_eval:n { #1 - 1 } .
31101     }
31102     #2
31103 }
31104 \cs_new:Npn \__fp_function_arg_few:w #1 @ { \exp_after:wN \c_nan_fp }
31105 \cs_new:Npn \__fp_function_arg_get:w #1#2#3\__fp_sep: #4 @
31106 {
31107     \__fp_exp_after_array_f:w #3\__fp_sep: \s__fp_expr_stop
31108     \exp_after:wN \exp_stop_f:
31109 }

```

(End of definition for \fp_set_function:nnn and others. This function is documented on page 275.)

```

\fp_clear_function:n 31110 \cs_new_protected:Npn \fp_clear_function:n #1
\__fp_clear_function:n 31111 { \exp_args:No \__fp_clear_function:n { \tl_to_str:n {#1} } }
31112 \cs_new_protected:Npn \__fp_clear_function:n #1
31113 {
31114     \__fp_id_if_invalid:nTF {#1}
31115     { \msg_error:nnn { fp } { id-invalid } {#1} }
31116     {
31117         \cs_set_eq:cN { __fp_#1_o:w } \tex_undefine:D
31118         \__fp_function_set_parsing:Nn \cs_set_eq:NN {#1}
31119     }
31120 }

```

(End of definition for \fp_clear_function:n and __fp_clear_function:n. This function is documented on page 275.)

```

31121 \code

```

Chapter 88

l3fparray implementation

```
31122 <*code>
```

```
31123 <@@=fp>
```

In analogy to l3intarray it would make sense to have <@@=fparray>, but we need direct access to `\__fp_parse:n` from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

88.1 Allocating arrays

There are somewhat more than $(2^{31} - 1)^2$ floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

`\g__fp_array_int` Used to generate unique names for the three integer arrays.

```
31124 \int_new:N \g__fp_array_int
```

(End of definition for \g__fp_array_int.)

`\l__fp_array_loop_int` Used to loop in `\__fp_array_gzero:N`.

```
31125 \int_new:N \l__fp_array_loop_int
```

(End of definition for \l__fp_array_loop_int.)

`\fparray_new:Nn`

`\fparray_new:cn`

`\__fp_array_new:nNNN`

Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

```
31126 \cs_new_protected:Npn \fparray_new:Nn #1#2
```

```
31127 {
```

```
31128   \tl_new:N #1
```

```
31129   \prg_replicate:nn { 3 }
```

```
31130   {
```

```
31131     \int_gincr:N \g__fp_array_int
```

```
31132     \exp_args:NNc \tl_gput_right:Nn #1
```

```
31133     { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
```

```
31134   }
```

```
31135 \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
```

```
31136 { \int_eval:n {#2} } #1 #1
```



```

31137 }
31138 \cs_generate_variant:Nn \fparray_new:Nn { c }
31139 \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
31140 {
31141   \int_compare:nNnTF {#1} < 0
31142   {
31143     \msg_error:nnn { kernel } { negative-array-size } {#1}
31144     \cs_undefine:N #1
31145     \int_gsub:Nn \g__fp_array_int { 3 }
31146   }
31147   {
31148     \intarray_new:Nn #2 {#1}
31149     \intarray_new:Nn #3 {#1}
31150     \intarray_new:Nn #4 {#1}
31151   }
31152 }

```

(End of definition for `\fparray_new:Nn` and `\__fp_array_new:nNNNN`. This function is documented on page 290.)

`\fparray_count:N` Size of any of the intarrays, here we pick the third.

```

\fparray_count:c
31153 \cs_new:Npn \fparray_count:N #1
31154 {
31155   \exp_after:wN \use_i:nnn
31156   \exp_after:wN \intarray_count:N #1
31157 }
31158 \cs_generate_variant:Nn \fparray_count:N { c }

```

(End of definition for `\fparray_count:N`. This function is documented on page 291.)

88.2 Array items

`\__fp_array_bounds:NNnTF` See the `\intarray` analogue: only names change. The functions `\fparray_gset:Nnn` and `\__fp_array_bounds_error:NNn` `\fparray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

31159 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
31160 {
31161   \if_int_compare:w 1 > #3 \exp_stop_f:
31162     \__fp_array_bounds_error:NNn #1 #2 {#3}
31163     #5
31164   \else:
31165     \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
31166       \__fp_array_bounds_error:NNn #1 #2 {#3}
31167       #5
31168     \else:
31169       #4
31170     \fi:
31171   \fi:
31172 }
31173 \cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
31174 {
31175   #1 { kernel } { out-of-bounds }
31176   { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
31177 }

```

(End of definition for _fp_array_bounds:NNnTF and _fp_array_bounds_error:NNn.)

\fparray_gset:Nnn Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next,
\fparray_gset:cnn and 8 trailing digits in the last.

```

\_fp_array_gset:NNNNww 31178 \cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
\_fp_array_gset:w 31179 {
\_fp_array_gset_recover:Nw 31180 \exp_after:wN \exp_after:wN
\_fp_array_gset_special:nnNNN 31181 \exp_after:wN \_fp_array_gset:NNNNww
\_fp_array_gset_normal:w 31182 \exp_after:wN #1
31183 \exp_after:wN #1
31184 \int_value:w \int_eval:n {#2} \exp_after:wN \_fp_sep:
31185 \exp:w \exp_end_continue_f:w \_fp_parse:n {#3}
31186 }
31187 \cs_generate_variant:Nn \fparray_gset:Nnn { c }
31188 \cs_new_protected:Npn \_fp_array_gset:NNNNww #1#2#3#4#5 \_fp_sep: #6 \_fp_sep:
31189 {
31190 \_fp_array_bounds:NNnTF \msg_error:nneee #4 {#5}
31191 {
31192 \exp_after:wN \_fp_change_func_type:NNN
31193 \_fp_use_i_until_s:nw #6 \_fp_sep:
31194 \_fp_array_gset:w
31195 \_fp_array_gset_recover:Nw
31196 #6 \_fp_sep: {#5} #1 #2 #3
31197 }
31198 { }
31199 }
31200 \cs_new_protected:Npn \_fp_array_gset_recover:Nw #1#2 \_fp_sep:
31201 {
31202 \_fp_error:nffn { unknown-type } { \tl_to_str:n { #2 \_fp_sep: } } { } { }
31203 \exp_after:wN #1 \c_nan_fp
31204 }
31205 \cs_new_protected:Npn \_fp_array_gset:w \s_fp \_fp_chk:w #1#2
31206 {
31207 \if_case:w #1 \exp_stop_f:
31208 \_fp_case_return:nw { \_fp_array_gset_special:nnNNN {#2} }
31209 \or: \exp_after:wN \_fp_array_gset_normal:w
31210 \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { #2 3 } }
31211 \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { 1 } }
31212 \fi:
31213 \s_fp \_fp_chk:w #1 #2
31214 }
31215 \cs_new_protected:Npn \_fp_array_gset_normal:w
31216 \s_fp \_fp_chk:w 1 #1 #2 #3#4#5 \_fp_sep: #6#7#8#9
31217 {
31218 \_kernel_intarray_gset:Nnn #7 {#6} {#2}
31219 \_kernel_intarray_gset:Nnn #8 {#6}
31220 { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
31221 \_kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
31222 }
31223 \cs_new_protected:Npn \_fp_array_gset_special:nnNNN #1#2#3#4#5
31224 {
31225 \_kernel_intarray_gset:Nnn #3 {#2} {#1}
31226 \_kernel_intarray_gset:Nnn #4 {#2} {0}
31227 \_kernel_intarray_gset:Nnn #5 {#2} {0}

```

```
31228 }
```

(End of definition for \fpararray_gset:Nnn and others. This function is documented on page 290.)

\fpararray_gzero:N

\fpararray_gzero:c

```
31229 \cs_new_protected:Npn \fpararray_gzero:N #1
31230 {
31231   \int_zero:N \l__fp_array_loop_int
31232   \prg_replicate:nn { \fpararray_count:N #1 }
31233   {
31234     \int_incr:N \l__fp_array_loop_int
31235     \exp_after:wN \__fp_array_gset_special:nnNNN
31236     \exp_after:wN 0
31237     \exp_after:wN \l__fp_array_loop_int
31238     #1
31239   }
31240 }
31241 \cs_generate_variant:Nn \fpararray_gzero:N { c }
```

(End of definition for \fpararray_gzero:N. This function is documented on page 290.)

\fpararray_item:Nn

\fpararray_item:cn

\fpararray_item_to_tl:Nn

\fpararray_item_to_tl:cn

__fp_array_item:NwN

__fp_array_item:NNNnN

__fp_array_item:N

__fp_array_item:w

__fp_array_item_special:w

__fp_array_item_normal:w

```
31242 \cs_new:Npn \fpararray_item:Nn #1#2
31243 {
31244   \exp_after:wN \__fp_array_item:NwN
31245   \exp_after:wN #1
31246   \int_value:w \int_eval:n {#2} \__fp_sep:
31247   \__fp_to_decimal:w
31248 }
31249 \cs_generate_variant:Nn \fpararray_item:Nn { c }
31250 \cs_new:Npn \fpararray_item_to_tl:Nn #1#2
31251 {
31252   \exp_after:wN \__fp_array_item:NwN
31253   \exp_after:wN #1
31254   \int_value:w \int_eval:n {#2} \__fp_sep:
31255   \__fp_to_tl:w
31256 }
31257 \cs_generate_variant:Nn \fpararray_item_to_tl:Nn { c }
31258 \cs_new:Npn \__fp_array_item:NwN #1#2 \__fp_sep: #3
31259 {
31260   \__fp_array_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
31261   { \exp_after:wN \__fp_array_item:NNNnN #1 {#2} #3 }
31262   { \exp_after:wN #3 \c_nan_fp }
31263 }
31264 \cs_new:Npn \__fp_array_item:NNNnN #1#2#3#4
31265 {
31266   \exp_after:wN \__fp_array_item:N
31267   \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN \__fp_sep:
31268   \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN \__fp_sep:
31269   \int_value:w \__kernel_intarray_item:Nn #1 {#4} \__fp_sep:
31270 }
31271 \cs_new:Npn \__fp_array_item:N #1
31272 {
31273   \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
31274   \__fp_array_item:w #1
```

```

31275 }
31276 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 \__fp_sep: 1 #7 \__fp_sep:
31277 {
31278   \exp_after:wN \__fp_array_item_normal:w
31279   \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
31280   #7 \__fp_sep: {#2#3#4#5} {#6} \__fp_sep:
31281 }
31282 \cs_new:Npn \__fp_array_item_special:w #1 \__fp_sep: #2 \__fp_sep: #3 \__fp_sep: #4
31283 {
31284   \exp_after:wN #4
31285   \exp:w \exp_end_continue_f:w
31286   \if_case:w #3 \exp_stop_f:
31287     \exp_after:wN \c_zero_fp
31288   \or: \exp_after:wN \c_nan_fp
31289   \or: \exp_after:wN \c_minus_zero_fp
31290   \or: \exp_after:wN \c_inf_fp
31291   \else: \exp_after:wN \c_minus_inf_fp
31292   \fi:
31293 }
31294 \cs_new:Npn \__fp_array_item_normal:w
31295   #1 #2#3#4#5 #6 \__fp_sep: #7 \__fp_sep: #8 \__fp_sep: #9
31296   { #9 \s__fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} \__fp_sep: }

```

(End of definition for \fpararray_item:Nn and others. These functions are documented on page 291.)

\fpararray_if_exist_p:N
 \fpararray_if_exist_p:c
 \fpararray_if_exist:NTF
 \fpararray_if_exist:cTF

Copies of the cs functions defined in l3basics.

```

31297 \prg_new_eq_conditional:NNn \fpararray_if_exist:N \cs_if_exist:N
31298   { TF , T , F , p }
31299 \prg_new_eq_conditional:NNn \fpararray_if_exist:c \cs_if_exist:c
31300   { TF , T , F , p }

```

(End of definition for \fpararray_if_exist:NTF. This function is documented on page 291.)

31301 </code>

Chapter 89

l3bitset implementation

```
31302 <*code>
31303 <@@=bitset>
```

A `bitset` is a string variable.

```
\bitset_new:N
\bitset_new:c 31304 \cs_new_protected:Npn \bitset_new:N #1
\bitset_new:Nn 31305 {
\bitset_new:cn 31306   \__kernel_chk_if_free_cs:N #1
31307   \cs_gset_eq:NN #1 \c_zero_str
31308   \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
31309 }
31310 \cs_new_protected:Npn \bitset_new:Nn #1 #2
31311 {
31312   \__kernel_chk_if_free_cs:N #1
31313   \cs_gset_eq:NN #1 \c_zero_str
31314   \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
31315   \prop_gset_from_keyval:cn
31316     { g__bitset_ \cs_to_str:N #1 _name_prop }
31317   {#2}
31318 }
31319 \cs_generate_variant:Nn \bitset_new:N { c }
31320 \cs_generate_variant:Nn \bitset_new:Nn { c }
```

(End of definition for `\bitset_new:N` and `\bitset_new:Nn`. These functions are documented on page 293.)

```
\bitset_addto_named_index:Nn
31321 \cs_new_protected:Npn \bitset_addto_named_index:Nn #1#2
31322 {
31323   \prop_gput_from_keyval:cn
31324     { g__bitset_ \cs_to_str:N #1 _name_prop } { #2 }
31325 }
```

(End of definition for `\bitset_addto_named_index:Nn`. This function is documented on page 293.)

```
\bitset_if_exist_p:N Existence tests.
\bitset_if_exist_p:c 31326 \prg_new_eq_conditional:NNn
\bitset_if_exist:NTF 31327 \bitset_if_exist:N \str_if_exist:N { p , T , F , TF }
\bitset_if_exist:cTF
```

```

31328 \prg_new_eq_conditional:Nn
31329 \bitset_if_exist:c \str_if_exist:c { p , T , F , TF }

```

(End of definition for \bitset_if_exist:NTF. This function is documented on page 294.)

```

\__bitset_set_true:Nn
\__bitset_gset_true:Nn
\__bitset_set_false:Nn
\__bitset_gset_false:Nn
\__bitset_set:Nnn

```

The internal command uses only numbers (integer expressions) for the position. A bit is set by either extending the string or by splitting it and then inserting an 1. It is not checked if the value was already 1.

```

31330 \cs_new_protected:Npn \__bitset_set_true:Nn #1#2
31331 { \__bitset_set:NnnN \str_set:Ne #1 {#2} 1 }
31332 \cs_new_protected:Npn \__bitset_gset_true:Nn #1#2
31333 { \__bitset_set:NnnN \str_gset:Ne #1 {#2} 1 }
31334 \cs_new_protected:Npn \__bitset_set_false:Nn #1#2
31335 { \__bitset_set:NnnN \str_set:Ne #1 {#2} 0 }
31336 \cs_new_protected:Npn \__bitset_gset_false:Nn #1#2
31337 { \__bitset_set:NnnN \str_gset:Ne #1 {#2} 0 }
31338 \cs_new_protected:Npn \__bitset_set:NnnN #1#2#3#4
31339 {
31340   \int_compare:nNnT {#3} > { 0 }
31341   {
31342     \int_compare:nNnTF { \str_count:N #2 } < {#3}
31343     {
31344       #1 #2
31345       {
31346         #4
31347         \prg_replicate:nn { #3 - \str_count:N #2 - 1 } { 0 }
31348         #2
31349       }
31350     }
31351     {
31352       #1 #2
31353       {
31354         \str_range:Nnn #2 { 1 } { -1 - (#3) }
31355         #4
31356         \str_range:Nnn #2 { 1 - (#3) } { -1 }
31357       }
31358     }
31359   }
31360 }

```

(End of definition for __bitset_set_true:Nn and others.)

```

\l__bitset_tmp_int

```

```

31361 \int_new:N \l__bitset_tmp_int

```

(End of definition for \l__bitset_tmp_int.)

```

\__bitset_test_digits:nTF
\__bitset_test_digits_end:n
\__bitset_test_digits:w

```

<https://chat.stackexchange.com/transcript/message/56878159#56878159>

```

31362 \prg_new_protected_conditional:Npnn \__bitset_test_digits:n #1 { TF }
31363 {
31364   \tex_afterassignment:D \__bitset_test_digits:w
31365   \l__bitset_tmp_int = 0 \tl_trim_spaces_apply:nN {#1} \tl_to_str:n
31366   \__bitset_test_digits_end:
31367   \use_i:nnn \if_false:
31368   \__bitset_test_digits_end:

```

```

31369     \if_int_compare:w \c_zero_int < \l__bitset_tmp_int
31370     \prg_return_true:
31371     \else:
31372     \prg_return_false:
31373     \fi:
31374 }
31375 \cs_new_eq:NN \__bitset_test_digits_end: \exp_stop_f:
31376 \cs_new_protected:Npn \__bitset_test_digits:w #1 \__bitset_test_digits_end: { }

(End of definition for \__bitset_test_digits:nTF, \__bitset_test_digits_end:n, and \__bitset-
test_digits:w.)

```

The user commands must first translate the argument to an index number.

```

\bitset_set_true:Nn
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn
\bitset_set_false:Nn
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn
\__bitset_set_aux:NNn
31377 \cs_new_protected:Npn \bitset_set_true:Nn #1#2
31378 { \__bitset_set:NnN \__bitset_set_true:Nn #1 {#2} }
31379 \cs_new_protected:Npn \bitset_gset_true:Nn #1#2
31380 { \__bitset_set:NnN \__bitset_gset_true:Nn #1 {#2} }
31381 \cs_new_protected:Npn \bitset_set_false:Nn #1#2
31382 { \__bitset_set:NnN \__bitset_set_false:Nn #1 {#2} }
31383 \cs_new_protected:Npn \bitset_gset_false:Nn #1#2
31384 { \__bitset_set:NnN \__bitset_gset_false:Nn #1 {#2} }
31385 \cs_new_protected:Npn \__bitset_set:NnN #1#2#3
31386 {
31387     \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
31388     {
31389         #1 #2
31390         {
31391             \prop_item:cn { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
31392         }
31393     }
31394     {
31395         \__bitset_test_digits:nTF {#3}
31396         {
31397             #1 #2 {#3}
31398             \prop_gput:cn { g__bitset_ \cs_to_str:N #2 _name_prop } {#3} {#3}
31399         }
31400         {
31401             \msg_warning:nnee { bitset } { unknown-name }
31402             { \token_to_str:N #2 }
31403             { \tl_to_str:n {#3} }
31404         }
31405     }
31406 }
31407 \cs_generate_variant:Nn \bitset_set_true:Nn { c }
31408 \cs_generate_variant:Nn \bitset_gset_true:Nn { c }
31409 \cs_generate_variant:Nn \bitset_set_false:Nn { c }
31410 \cs_generate_variant:Nn \bitset_gset_false:Nn { c }

```

(End of definition for \bitset_set_true:Nn and others. These functions are documented on page 294.)

```

\bitset_clear:N
\bitset_clear:c
\bitset_gclear:N
\bitset_gclear:c
31411 \cs_new_protected:Npn \bitset_clear:N #1
31412 {
31413     \str_set_eq:NN #1 \c_zero_str
31414 }

```

```

31415 \cs_new_protected:Npn \bitset_gclear:N #1
31416 {
31417   \str_gset_eq:NN #1 \c_zero_str
31418 }
31419 \cs_generate_variant:Nn \bitset_clear:N { c }
31420 \cs_generate_variant:Nn \bitset_gclear:N { c }

```

(End of definition for `\bitset_clear:N` and `\bitset_gclear:N`. These functions are documented on page 294.)

`\bitset_to_arabic:N` The naming of the commands follow the names in the `int` module. `\bitset_to_arabic:N` uses `\int_from_bin:n` if the string is shorter than 32 and the slower `\fp_eval:n` for larger bitsets.

```

\bitset_to_arabic:c
\bitset_to_bin:N
\bitset_to_bin:c
\__bitset_to_int:nN
31421 \cs_new:Npn \bitset_to_arabic:N #1
31422 {
31423   \int_compare:nNnTF { \str_count:N #1 } < { 32 }
31424   { \exp_args:No \int_from_bin:n {#1} }
31425   {
31426     \exp_after:wN \__bitset_to_int:nN \exp_after:wN 0
31427     #1 \q_recursion_tail \q_recursion_stop
31428   }
31429 }
31430 \cs_new:Npn \__bitset_to_int:nN #1#2
31431 {
31432   \quark_if_recursion_tail_stop_do:Nn #2 {#1}
31433   \exp_args:Nf \__bitset_to_int:nN { \fp_eval:n { #1 * 2 + #2 } }
31434 }
31435 \cs_new:Npn \bitset_to_bin:N #1
31436 {
31437   #1
31438 }
31439 \cs_generate_variant:Nn \bitset_to_arabic:N { c }
31440 \cs_generate_variant:Nn \bitset_to_bin:N { c }

```

(End of definition for `\bitset_to_arabic:N`, `\bitset_to_bin:N`, and `\__bitset_to_int:nN`. These functions are documented on page 295.)

`\bitset_use:N`

```

\bitset_use:c
31441 \cs_new_eq:NN \bitset_use:N \tl_use:N
31442 \cs_generate_variant:Nn \bitset_use:N { c }

```

(End of definition for `\bitset_use:N`. This function is documented on page 295.)

`\bitset_item:Nn` All bits that have been set at anytime have an entry in the prop, so we can take everything else as 0.

```

\bitset_item:cn
31443 \cs_new:Npn \bitset_item:Nn #1#2
31444 {
31445   \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #1 _name_prop } {#2}
31446   {
31447     \int_eval:n
31448     {
31449       \str_item:Nn #1
31450       { 0 - ( \prop_item:cn { g__bitset_ \cs_to_str:N #1 _name_prop } {#2} ) }
31451       +0
31452     }

```



```

31453     }
31454     {
31455         0
31456     }
31457 }
31458 \cs_generate_variant:Nn \bitset_item:Nn { c }

```

(End of definition for \bitset_item:Nn. This function is documented on page 294.)

\bitset_show:N

```

\bitset_show:c 31459 \cs_new_protected:Npn \bitset_show:N { \__bitset_show:NN \msg_show:nneeee }
\bitset_log:N 31460 \cs_generate_variant:Nn \bitset_show:N { c }
\bitset_log:c 31461 \cs_new_protected:Npn \bitset_log:N { \__bitset_show:NN \msg_log:nneeee }
31462 \cs_generate_variant:Nn \bitset_log:N { c }
31463 \cs_new_protected:Npn \__bitset_show:NN #1#2
31464 {
31465     \__kernel_chk_defined:NT #2
31466     {
31467         #1 { bitset } { show }
31468         { \token_to_str:N #2 }
31469         { \bitset_to_bin:N #2 }
31470         { \bitset_to_arabic:N #2 }
31471         { }
31472     }
31473 }

```

(End of definition for \bitset_show:N and \bitset_log:N. These functions are documented on page 295.)

\bitset_show_named_index:N

```

\bitset_show_named_index:c 31474 \cs_new_protected:Npn \bitset_show_named_index:N
\bitset_log_named_index:N 31475 { \__bitset_show_named_index:NN \msg_show:nneeee }
\bitset_log_named_index:c 31476 \cs_generate_variant:Nn \bitset_show_named_index:N { c }
31477 \cs_new_protected:Npn \bitset_log_named_index:N
31478 { \__bitset_show_named_index:NN \msg_log:nneeee }
31479 \cs_generate_variant:Nn \bitset_log_named_index:N { c }
31480 \cs_new_protected:Npn \__bitset_show_named_index:NN #1#2
31481 {
31482     \__kernel_chk_defined:NT #2
31483     {
31484         #1 { bitset } { show-names }
31485         { \token_to_str:N #2 }
31486         { \prop_map_function:cN { g__bitset_ \cs_to_str:N #2 _name_prop } \msg_show_item:
31487           { } { } }
31488     }
31489 }

```

(End of definition for \bitset_show_named_index:N and \bitset_log_named_index:N. These functions are documented on page 295.)

89.1 Messages

```

31490 \msg_new:nnn { bitset } { show }
31491 {

```

```

31492     The~bitset~#1~has~the~representation: \\
31493     >~binary:~#2  \\
31494     >~arabic:~#3  .
31495   }
31496 \msg_new:nnn { bitset } { show-names }
31497   {
31498     The~bitset~#1~
31499     \tl_if_empty:nTF {#2}
31500       { knows~no~names~yet \\>~ . }
31501       { knows~the~name/index~pairs~(without~outer~braces): #2 . }
31502   }
31503 \msg_new:nnn { bitset } { unknown-name }
31504   { The~name~'#2'~is~unknown~for~bitset~\tl_to_str:n {#1} }
31505 \prop_gput:Nnn \g_msg_module_name_prop { bitset } { LaTeX }
31506 \prop_gput:Nnn \g_msg_module_type_prop { bitset } { }
31507 </code>

```

Chapter 90

l3cctab implementation

```
31508 <*code>
31509 <@@=cctab>
```

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future XeTeX may add support, at which point the conditionals below would be different.)

90.1 Variables

<pre>\g__cctab_stack_seq \g__cctab_unused_seq</pre>	List of catcode tables saved by nested <code>\cctab_begin:N</code>, to restore catcodes at the matching <code>\cctab_end:.</code> When popped from the <code>\g__cctab_stack_seq</code> the table numbers are stored in <code>\g__cctab_unused_seq</code> for later reuse. <pre>31510 \seq_new:N \g__cctab_stack_seq 31511 \seq_new:N \g__cctab_unused_seq</pre> <i>(End of definition for \g__cctab_stack_seq and \g__cctab_unused_seq.)</i>
<pre>\g__cctab_group_seq</pre>	A stack to store the group level when a catcode table started. <pre>31512 \seq_new:N \g__cctab_group_seq</pre> <i>(End of definition for \g__cctab_group_seq.)</i>
<pre>\g__cctab_allocate_int</pre>	Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement <code>\cctab_new:N</code>. In other engines it is used to make csnames for dynamic tables. <pre>31513 \int_new:N \g__cctab_allocate_int</pre> <i>(End of definition for \g__cctab_allocate_int.)</i>
<pre>\l__cctab_tmpa_tl \l__cctab_tmpp_tl</pre>	Scratch space. For instance, when popping <code>\g__cctab_stack_seq/\g__cctab_unused_seq</code>, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines. <pre>31514 \tl_new:N \l__cctab_tmpa_tl 31515 \tl_new:N \l__cctab_tmpp_tl</pre> <i>(End of definition for \l__cctab_tmpa_tl and \l__cctab_tmpp_tl.)</i>

`\g__cctab_endlinechar_prop` In LuaTeX we store the `\endlinechar` associated to each `\catcodetable` in a property list, unless it is the default value 13.

```
31516 \prop_new:N \g__cctab_endlinechar_prop
```

(End of definition for `\g__cctab_endlinechar_prop`.)

90.2 Allocating category code tables

`\cctab_new:N` The `\__cctab_new:N` auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in `\cctab_new:N`, which sets catcodes to `iniTeX` values, and in `\cctab_begin:N/\cctab_end:` for dynamically allocated tables.

`\__cctab_new:N` First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, `\newcatcodetable` does not include the initialization, so that is added explicitly.

`\__cctab_gstore:Nnn`

```
31517 \sys_if_engine luatex:TF
31518 {
31519   \cs_new_protected:Npn \cctab_new:N #1
31520   {
31521     \__kernel_chk_if_free_cs:N #1
31522     \__cctab_new:N #1
31523   }
31524   \cs_new_protected:Npn \__cctab_new:N #1
31525   {
31526     \newcatcodetable #1
31527     \tex_initcatcodetable:D #1
31528   }
31529 }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with `iniTeX` codes. The `\debug_suspend:` and `\debug_resume:` functions prevent errors and logging from `debug` commands which are either duplicate or false when `\__cctab_new:N` is used by `\cctab_new:N` or `\cctab_const:Nn`. The index base is out-by-one, so we have an internal function to handle that. The `iniTeX` `\endlinechar` is 13.

```
31530 {
31531   \cs_new_protected:Npn \__cctab_new:N #1
31532   {
31533     \debug_suspend:
31534     \intarray_new:Nn #1 { 257 }
31535     \debug_resume:
31536   }
31537   \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
31538   { \intarray_gset:Nnn #1 { #2 + 1 } {#3} }
31539   \cs_new_protected:Npn \cctab_new:N #1
31540   {
31541     \__kernel_chk_if_free_cs:N #1
31542     \__cctab_new:N #1
31543     \int_step_inline:nn { 256 }
31544     { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
31545     \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
31546     \__cctab_gstore:Nnn #1 { 0 } { 9 }
31547     \__cctab_gstore:Nnn #1 { 13 } { 5 }
```

```

31548     \__cctab_gstore:Nnn #1 { 32 } { 10 }
31549     \__cctab_gstore:Nnn #1 { 37 } { 14 }
31550     \int_step_inline:nnn { 65 } { 90 }
31551     { \__cctab_gstore:Nnn #1 {##1} { 11 } }
31552     \__cctab_gstore:Nnn #1 { 92 } { 0 }
31553     \int_step_inline:nnn { 97 } { 122 }
31554     { \__cctab_gstore:Nnn #1 {##1} { 11 } }
31555     \__cctab_gstore:Nnn #1 { 127 } { 15 }
31556   }
31557 }
31558 \cs_generate_variant:Nn \cctab_new:N { c }

```

(End of definition for `\cctab_new:N`, `\__cctab_new:N`, and `\__cctab_gstore:Nnn`. This function is documented on page 296.)

90.3 Saving category code tables

```

\__cctab_gset:n
\__cctab_gset_aux:n

```

In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

31559 \sys_if_engine luatex:TF
31560 {
31561   \cs_new_protected:Npn \__cctab_gset:n #1
31562   { \exp_args:Nf \__cctab_gset_aux:n { \int_eval:n {#1} } }
31563   \cs_new_protected:Npn \__cctab_gset_aux:n #1
31564   {
31565     \tex_savecatcodetable:D #1 \scan_stop:
31566     \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
31567     { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
31568     {
31569       \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
31570       \tex_endlinechar:D
31571     }
31572   }
31573 }
31574 {
31575   \cs_new_protected:Npn \__cctab_gset:n #1
31576   {
31577     \int_step_inline:nn { 256 }
31578     {
31579       \__kernel_intarray_gset:Nnn #1 {##1}
31580       { \char_value_catcode:n { ##1 - 1 } }
31581     }
31582     \__kernel_intarray_gset:Nnn #1 { 257 }
31583     { \tex_endlinechar:D }
31584   }
31585 }

```

(End of definition for `\__cctab_gset:n` and `\__cctab_gset_aux:n`.)

`\cctab_gset:Nn` Category code tables are always global, so only one version of assignments is needed.
`\cctab_gset:cn` Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

31586 \cs_new_protected:Npn \cctab_gset:Nn #1#2
31587 {
31588   \__cctab_chk_if_valid:NT #1
31589   {
31590     \group_begin:
31591       \cctab_select:N \c_initex_cctab
31592       #2 \scan_stop:
31593       \__cctab_gset:n {#1}
31594     \group_end:
31595   }
31596 }
31597 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End of definition for `\cctab_gset:Nn`. This function is documented on page 296.)

`\cctab_gsave_current:N` Very simple.

```

\cctab_gsave_current:c
31598 \cs_new_protected:Npn \cctab_gsave_current:N #1
31599 {
31600   \__cctab_chk_if_valid:NT #1
31601   { \__cctab_gset:n {#1} }
31602 }
31603 \cs_generate_variant:Nn \cctab_gsave_current:N { c }

```

(End of definition for `\cctab_gsave_current:N`. This function is documented on page 296.)

90.4 Using category code tables

`\g__cctab_tmp_cctab`
`\__cctab_tmp_cctab_name:`

In LuaTeX, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \char_set_catcode_other:N \_
  \cctab_end:
  \cctab_begin:N \c_document_cctab
  \int_compare:nTF { \char_value_catcode:n { ‘_’ } = 8 }
  { \TRUE } { \ERROR }
  \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong

category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```
\use:n
{
  \cctab_begin:N \c_document_cctab
  \group_begin:
    \cctab_begin:N \c_str_cctab
    \cctab_end:
  \group_end:
  \cctab_end:
}

31604 \sys_if_engine luatex:T
31605 {
31606   \__cctab_new:N \g__cctab_tmp_cctab
31607   \cs_new:Npn \__cctab_tmp_cctab_name:
31608   {
31609     g__cctab_tmp
31610     \tex_romannumeral:D \tex_currentgrouplevel:D
31611     _cctab
31612   }
31613 }
```

(End of definition for `\g__cctab_tmp_cctab` and `\__cctab_tmp_cctab_name:.`)

`\cctab_select:N` The public function simply checks the `\cctab var` exists before using the engine-dependent `\__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```
31614 \cs_new_protected:Npn \cctab_select:N #1
31615 { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
31616 \cs_generate_variant:Nn \cctab_select:N { c }
31617 \sys_if_engine luatex:TF
31618 {
31619   \cs_new_protected:Npn \__cctab_select:N #1
31620   {
31621     \tex_catcodetable:D #1
31622     \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_tmpa_tl
31623     { \int_set:Nn \tex_endlinechar:D { \l__cctab_tmpa_tl } }
31624     { \int_set:Nn \tex_endlinechar:D { 13 } }
31625     \cs_if_exist:cF { \__cctab_tmp_cctab_name: }
31626     { \exp_args:Nc \__cctab_new:N { \__cctab_tmp_cctab_name: } }
31627     \exp_args:Nc \tex_savecatcodetable:D { \__cctab_tmp_cctab_name: }
31628     \exp_args:Nc \tex_catcodetable:D { \__cctab_tmp_cctab_name: }
31629   }
31630 }
31631 {
31632   \cs_new_protected:Npn \__cctab_select:N #1
31633   {
31634     \int_step_inline:nn { 256 }
31635     {
```

```

31636         \char_set_catcode:nn { ##1 - 1 }
31637         { \__kernel_intarray_item:Nn #1 {##1} }
31638     }
31639     \int_set:Nn \tex_endlinechar:D
31640     { \__kernel_intarray_item:Nn #1 { 257 } }
31641 }
31642 }

```

(End of definition for `\cctab_select:N` and `\__cctab_select:N`. This function is documented on page 297.)

`\g__cctab_next_cctab` For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `\__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_tmpa_tl`. In LuaTeX this simply calls `\__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end:`.

```

31643 \sys_if_engine luatex:TF
31644 {
31645     \cs_new_protected:Npn \__cctab_begin_aux:
31646     {
31647         \__cctab_new:N \g__cctab_next_cctab
31648         \tl_set:NW \l__cctab_tmpa_tl \g__cctab_next_cctab
31649         \cs_undefine:N \g__cctab_next_cctab
31650     }
31651 }
31652 {
31653     \cs_new_protected:Npn \__cctab_begin_aux:
31654     {
31655         \int_gincr:N \g__cctab_allocate_int
31656         \exp_args:Nc \__cctab_new:N
31657         { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
31658         \exp_args:NNc \tl_set:Nn \l__cctab_tmpa_tl
31659         { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
31660     }
31661 }

```

(End of definition for `\g__cctab_next_cctab` and `\__cctab_begin_aux:`.)

`\cctab_begin:N` Check the `<cctab var>` exists, to avoid low-level errors. Get in `\l__cctab_tmpa_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `\__cctab_begin_aux:` if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_tmpa_tl` and save that table number in a stack before selecting the desired catcodes.

```

31662 \cs_new_protected:Npn \cctab_begin:N #1
31663 {
31664     \__cctab_chk_if_valid:NT #1
31665     {
31666         \seq_gpop:NnF \g__cctab_unused_seq \l__cctab_tmpa_tl
31667         { \__cctab_begin_aux: }
31668     }
31669     \__cctab_chk_group_begin:e
31670     { \__cctab_nesting_number:N \l__cctab_tmpa_tl }

```



```

31670         \seq_gpush:NV \g__cctab_stack_seq \l__cctab_tmpa_tl
31671         \exp_args:NV \__cctab_gset:n \l__cctab_tmpa_tl
31672         \__cctab_select:N #1
31673     }
31674 }
31675 \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End of definition for `\cctab_begin:N`. This function is documented on page 297.)

\cctab_end: Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_tmpa_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```

31676 \cs_new_protected:Npn \cctab_end:
31677 {
31678     \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_tmpa_tl
31679     {
31680         \seq_gpush:NV \g__cctab_unused_seq \l__cctab_tmpa_tl
31681         \exp_args:Ne \__cctab_chk_group_end:n
31682         { \__cctab_nesting_number:N \l__cctab_tmpa_tl }
31683         \__cctab_select:N \l__cctab_tmpa_tl
31684     }
31685     { \msg_error:nn { cctab } { extra-end } }
31686 }

```

(End of definition for `\cctab_end:`. This function is documented on page 297.)

`\__cctab_chk_group_begin:n` Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding $\TeX$ groups. `\__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `\__cctab_group_<cctab-level>_chk:`.

`\__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end:`.

```

\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `\__cctab_group_<cctab-level>_chk:`, and then check if it exist in `\cctab_end:`. If it doesn't, we know there was a group end where it shouldn't.

The `<cctab-level>` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `\__cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `\cctab_level` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

31687 \cs_new_protected:Npn \__cctab_chk_group_begin:n #1
31688 {
31689   \seq_gpush:N \g__cctab_group_seq
31690   { \int_use:N \tex_currentgrouplevel:D }
31691   \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
31692 }
31693 \cs_generate_variant:Nn \__cctab_chk_group_begin:n { e }
31694 \cs_new_protected:Npn \__cctab_chk_group_end:n #1
31695 {
31696   \seq_gpop:NN \g__cctab_group_seq \l__cctab_tmpb_tl
31697   \bool_lazy_and:nnF
31698   {
31699     \int_compare_p:nNn
31700     { \tex_currentgrouplevel:D } = { \l__cctab_tmpb_tl }
31701   }
31702   { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
31703   {
31704     \msg_error:nne { cctab } { group-mismatch }
31705     {
31706       \int_sign:n
31707       { \tex_currentgrouplevel:D - \l__cctab_tmpb_tl }
31708     }
31709   }
31710   \cs_undefine:c { __cctab_group_ #1 _chk: }
31711 }

```

(End of definition for `\__cctab_chk_group_begin:n` and `\__cctab_chk_group_end:n`.)

```

\__cctab_nesting_number:N
\__cctab_nesting_number:w

```

This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.

```

31712 \sys_if_engine luatex:TF
31713 { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
31714 {
31715   \cs_new:Npn \__cctab_nesting_number:N #1
31716   {
31717     \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
31718     \exp_after:wN \token_to_str:N #1
31719   }
31720   \use:e
31721   {
31722     \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
31723     #1 \tl_to_str:n { g__cctab_ } #2 \tl_to_str:n { _cctab } {#2}
31724   }
31725 }

```

(End of definition for `\__cctab_nesting_number:N` and `\__cctab_nesting_number:w`.)

Finally, install some code at the end of the $\TeX$ run to check that all `\cctab_begin:N` were ended by some `\cctab_end:.`

```

31726 \cs_if_exist:NT \hook_gput_code:nnn
31727 {
31728   \hook_gput_code:nnn { enddocument/end } { cctab }
31729   {
31730     \seq_if_empty:NF \g__cctab_stack_seq
31731     { \msg_error:nn { cctab } { missing-end } }
31732   }
31733 }

```

`\cctab_item:Nn` Evaluate the integer argument only once. In most engines the `cctab` variable only has 256 entries so we only look up the catcode for these entries, otherwise we use the current catcode. In particular, for out-of-range values we use whatever fall-back `\char_value_catcode:n`. In Lua $\TeX$, we use the `tex.getcatcode` function.

`\cctab_item:cn`

```

31734 \cs_new:Npn \cctab_item:Nn #1#2
31735 { \exp_args:Nf \__cctab_item:nN { \int_eval:n {#2} } #1 }
31736 \sys_if_engine luatex:TF
31737 {
31738   \cs_new:Npn \__cctab_item:nN #1#2
31739   { \lua_now:e { tex.print(-2, tex.getcatcode(\int_use:N #2, #1)) } }
31740 }
31741 {
31742   \cs_new:Npn \__cctab_item:nN #1#2
31743   {
31744     \int_compare:nNnTF {#1} < { 256 }
31745     { \intarray_item:Nn #2 { #1 + 1 } }
31746     { \char_value_catcode:n {#1} }
31747   }
31748 }
31749 \cs_generate_variant:Nn \cctab_item:Nn { c }

```

(End of definition for `\cctab_item:Nn`. This function is documented on page 297.)

90.5 Category code table conditionals

`\cctab_if_exist_p:N` Checks whether a $\langle cctab\ var \rangle$ is defined.

`\cctab_if_exist_p:c`

`\cctab_if_exist:NTF`

`\cctab_if_exist:cTF`

```

31750 \prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
31751 { TF , T , F , p }
31752 \prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
31753 { TF , T , F , p }

```

(End of definition for \cctab_if_exist:NTF. This function is documented on page 297.)

```

\__cctab_chk_if_valid:NTF
\__cctab_chk_if_valid_aux:NTF

```

Checks whether the argument is defined and whether it is a valid `<cctab var>`. In LuaTeX the validity of the `<cctab var>` is checked by the engine, which complains if the argument is not a `\chardef`'ed constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the cmr10 font).

```

31754 \prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
31755 { TF , T , F }
31756 {
31757   \cctab_if_exist:NTF #1
31758   {
31759     \__cctab_chk_if_valid_aux:NTF #1
31760     { \prg_return_true: }
31761     {
31762       \msg_error:nne { cctab } { invalid-cctab }
31763       { \token_to_str:N #1 }
31764       \prg_return_false:
31765     }
31766   }
31767   {
31768     \msg_error:nne { kernel } { command-not-defined }
31769     { \token_to_str:N #1 }
31770     \prg_return_false:
31771   }
31772 }
31773 \sys_if_engine luatex:TF
31774 {
31775   \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
31776   {
31777     \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
31778   }
31779   \cs_if_exist:NT \c_syst_catcodes_n
31780   {
31781     \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
31782     {
31783       \int_compare:nTF { #1 <= \c_syst_catcodes_n }
31784     }
31785   }
31786 }
31787 {
31788   \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
31789   {
31790     \exp_args:Nf \str_if_in:nnTF
31791     { \cs_meaning:N #1 }
31792     { select-font~cmr10~at~ }
31793   }
31794 }

```

(End of definition for __cctab_chk_if_valid:NTF and __cctab_chk_if_valid_aux:NTF.)

90.6 Constant category code tables

`\cctab_const:Nn` Creates a new `\cctab var` then sets it with the `iniTeX` and user-supplied codes. To avoid false debug errors, we write out implementation of `\cctab_new:N` and `\cctab_gset:Nn` instead of directly using them here. The initialization part in `\cctab_new:N` in non-`LuaTeX` is omitted as it's covered by the `iniTeX` settings.

```
\cctab_const:cn
31795 \cs_new_protected:Npn \cctab_const:Nn #1#2
31796 {
31797   \__kernel_chk_if_free_cs:N #1
31798   \__cctab_new:N #1
31799   \group_begin:
31800     \cctab_select:N \c_initex_cctab
31801     #2 \scan_stop:
31802     \__cctab_gset:n {#1}
31803   \group_end:
31804 }
31805 \cs_generate_variant:Nn \cctab_const:Nn { c }
```

(End of definition for `\cctab_const:Nn`. This function is documented on page 296.)

`\c_initex_cctab` Creating category code tables means thinking starting from `iniTeX`. For all-other and **`\c_other_cctab`** the standard “string” tables that’s easy.

```
\c_str_cctab
31806 \cctab_new:N \c_initex_cctab
31807 \cctab_const:Nn \c_other_cctab
31808 {
31809   \cctab_select:N \c_initex_cctab
31810   \int_set:Nn \tex_endlinechar:D { -1 }
31811   \int_step_inline:nnn { 0 } { 127 }
31812     { \char_set_catcode_other:n {#1} }
31813 }
31814 \cctab_const:Nn \c_str_cctab
31815 {
31816   \cctab_select:N \c_other_cctab
31817   \char_set_catcode_space:n { 32 }
31818 }
```

(End of definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 298.)

`\c_code_cctab` To pick up document-level category codes, we need to delay set up to the end of the **`\c_document_cctab`** format, where that’s possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```
31819 \cs_if_exist:NTF \@expl@finalise@setup@@
31820 { \tl_gput_right:Nn \@expl@finalise@setup@@ }
31821 { \use:n }
31822 {
31823   \__cctab_new:N \c_code_cctab
31824   \group_begin:
31825     \int_set:Nn \tex_endlinechar:D { 32 }
31826     \char_set_catcode_invalid:n { 0 }
31827     \sys_if_engine_opentype:TF
31828       { \int_step_function:nN { 31 } \char_set_catcode_invalid:n }
```

```

31829     { \int_step_function:nnN { 31 } \char_set_catcode_active:n }
31830 \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
31831 \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
31832 \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
31833 \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
31834 \char_set_catcode_ignore:n      { 9 }   % tab
31835 \char_set_catcode_other:n       { 10 }  % lf
31836 \char_set_catcode_active:n      { 12 }  % ff
31837 \char_set_catcode_end_line:n    { 13 }  % cr
31838 \char_set_catcode_ignore:n      { 32 }  % space
31839 \char_set_catcode_parameter:n   { 35 }  % hash
31840 \char_set_catcode_math_toggle:n { 36 }  % dollar
31841 \char_set_catcode_comment:n     { 37 }  % percent
31842 \char_set_catcode_alignment:n   { 38 }  % ampersand
31843 \char_set_catcode_letter:n      { 58 }  % colon
31844 \char_set_catcode_escape:n      { 92 }  % backslash
31845 \char_set_catcode_math_superscript:n { 94 } % circumflex
31846 \char_set_catcode_letter:n      { 95 }  % underscore
31847 \char_set_catcode_group_begin:n { 123 } % left brace
31848 \char_set_catcode_other:n       { 124 } % pipe
31849 \char_set_catcode_group_end:n   { 125 } % right brace
31850 \char_set_catcode_space:n       { 126 } % tilde
31851 \char_set_catcode_invalid:n     { 127 } % ^^?
31852 \sys_if_engine_opentype:F
31853 { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
31854 \__cctab_gset:n { \c_code_cctab }
31855 \group_end:
31856 \cctab_const:Nn \c_document_cctab
31857 {
31858   \cctab_select:N \c_code_cctab
31859   \int_set:Nn \tex_endlinechar:D { 13 }
31860   \char_set_catcode_space:n      { 9 }
31861   \char_set_catcode_space:n      { 32 }
31862   \char_set_catcode_other:n      { 58 }
31863   \char_set_catcode_math_subscript:n { 95 }
31864   \char_set_catcode_active:n     { 126 }
31865 }
31866 }

```

(End of definition for `\c_code_cctab` and `\c_document_cctab`. These variables are documented on page 297.)

`\g_tmpa_cctab`
`\g_tmpb_cctab`

```

31867 \cctab_new:N \g_tmpa_cctab
31868 \cctab_new:N \g_tmpb_cctab

```

(End of definition for `\g_tmpa_cctab` and `\g_tmpb_cctab`. These variables are documented on page 298.)

90.7 Messages

```

31869 \msg_new:nnnn { cctab } { stack-full }
31870 { The~category~code~table~stack~is~exhausted. }
31871 {
31872   LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~

```

```

31873     but~there~is~no~more~space~to~do~this!
31874 }
31875 \msg_new:nnnn { cctab } { extra-end }
31876 { Extra~\iow_char:N\\cctab_end:~ignored~\msg_line_context:. }
31877 {
31878     LaTeX~came~across~a~\iow_char:N\\cctab_end:~without~a~matching~
31879     \iow_char:N\\cctab_begin:N.~This~command~will~be~ignored.
31880 }
31881 \msg_new:nnnn { cctab } { missing-end }
31882 { Missing~\iow_char:N\\cctab_end:~before~end~of~TeX~run. }
31883 {
31884     LaTeX~came~across~more~\iow_char:N\\cctab_begin:N~than~
31885     \iow_char:N\\cctab_end:.
31886 }
31887 \msg_new:nnnn { cctab } { invalid-cctab }
31888 { Invalid~\iow_char:N\\catcode~table. }
31889 {
31890     You~can~only~switch~to~a~\iow_char:N\\catcode~table~that~is~
31891     initialized~using~\iow_char:N\\cctab_new:N~or~
31892     \iow_char:N\\cctab_const:Nn.
31893 }
31894 \msg_new:nnnn { cctab } { group-mismatch }
31895 {
31896     \iow_char:N\\cctab_end:~occurred~in~a~
31897     \int_case:nn {#1}
31898     {
31899         { 0 } { different~group }
31900         { 1 } { higher~group~level }
31901         { -1 } { lower~group~level }
31902     } ~than~
31903     the~matching~\iow_char:N\\cctab_begin:N.
31904 }
31905 {
31906     Catcode~tables~and~groups~must~be~properly~nested,~but~
31907     you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~
31908     but~results~may~be~unexpected.
31909 }
31910 \prop_gput:Nnn \g_msg_module_name_prop { cctab } { LaTeX }
31911 \prop_gput:Nnn \g_msg_module_type_prop { cctab } { }
31912 \code

```

Chapter 91

Unicode implementation

```
31913 <*code>
31914 <@@=codepoint>
```

91.1 User functions

```
\codepoint_str_generate:n
  \_codepoint_str_generate:nnnn
\codepoint_generate:nn
\__codepoint_generate:nnnn
  \__codepoint_generate:n
```

Conversion of a codepoint to a character (Unicode engines) or to one or more bytes (8-bit engines) is required. For loading the data, all that is needed is the form which creates strings: these are outside the group as they will also be used when looking up data in the hash table storage at point-of-use. Later, we will also need functions that can generate character tokens for document use: those are defined below, in the data recovery setup.

```
31915 \sys_if_engine_opentype:TF
31916 {
31917   \cs_new:Npn \codepoint_str_generate:n #1
31918   {
31919     \int_compare:nNnTF {#1} = { '\ }
31920     { ~ }
31921     { \char_generate:nn {#1} { 12 } }
31922   }
31923   \cs_new:Npn \codepoint_generate:nn #1#2
31924   {
31925     \int_compare:nNnTF {#1} = { '\ }
31926     { ~ }
31927     {
31928       \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
31929       { \char_generate:nn {#1} {#2} }
31930     }
31931   }
31932 }
31933 {
31934   \cs_new:Npn \codepoint_str_generate:n #1
31935   {
31936     \int_compare:nNnTF {#1} = { '\ }
31937     { ~ }
31938     {
31939       \use:e
31940       {
31941         \exp_not:N \__codepoint_str_generate:nnnn
```



```

31942         \__kernel_codepoint_to_bytes:n {#1}
31943     }
31944 }
31945 }
31946 \cs_new:Npn \__codepoint_str_generate:nnnn #1#2#3#4
31947 {
31948     \char_generate:nn {#1} { 12 }
31949     \tl_if_blank:nF {#2}
31950     {
31951         \char_generate:nn {#2} { 12 }
31952         \tl_if_blank:nF {#3}
31953         {
31954             \char_generate:nn {#3} { 12 }
31955             \tl_if_blank:nF {#4}
31956             { \char_generate:nn {#4} { 12 } }
31957         }
31958     }
31959 }
31960 \cs_new:Npn \codepoint_generate:nn #1#2
31961 {
31962     \int_compare:nNnTF {#1} = { '\ }
31963     { ~ }
31964     {
31965         \int_compare:nNnTF {#1} < { "80 }
31966         {
31967             \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
31968             { \char_generate:nn {#1} {#2} }
31969         }
31970         {
31971             \use:e
31972             {
31973                 \exp_not:N \__codepoint_generate:nnnn
31974                 \__kernel_codepoint_to_bytes:n {#1}
31975             }
31976         }
31977     }
31978 }
31979 \cs_new:Npn \__codepoint_generate:nnnn #1#2#3#4
31980 {
31981     \__kernel_exp_not:w \exp_after:wN
31982     {
31983         \tex_expanded:D
31984         {
31985             \__codepoint_generate:n {#1}
31986             \__codepoint_generate:n {#2}
31987             \tl_if_blank:nF {#3}
31988             {
31989                 \__codepoint_generate:n {#3}
31990                 \tl_if_blank:nF {#4}
31991                 { \__codepoint_generate:n {#4} }
31992             }
31993         }
31994     }
31995 }

```

```

31996     \cs_new:Npn \__codepoint_generate:n #1
31997     {
31998         \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
31999         { \char_generate:nn {#1} { 13 } }
32000     }
32001 }

```

(End of definition for \codepoint_str_generate:n and others. These functions are documented on page 301.)

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

\__kernel_codepoint_to_bytes:n
\__codepoint_to_bytes_auxi:n
\__codepoint_to_bytes_auxii:Nnn
\__codepoint_to_bytes_auxiii:n
\__codepoint_to_bytes_outputi:nw
\__codepoint_to_bytes_outputii:nw
\__codepoint_to_bytes_outputiii:nw
\__codepoint_to_bytes_outputiv:nw
\__codepoint_to_bytes_output:nnn
\__codepoint_to_bytes_output:fnn
\__codepoint_to_bytes_end:
32002 \cs_new:Npn \__kernel_codepoint_to_bytes:n #1
32003 {
32004     \exp_args:Nf \__codepoint_to_bytes_auxi:n
32005     { \int_eval:n {#1} }
32006 }
32007 \cs_new:Npn \__codepoint_to_bytes_auxi:n #1
32008 {
32009     \if_int_compare:w #1 > "80 \exp_stop_f:
32010     \if_int_compare:w #1 < "800 \exp_stop_f:
32011         \__codepoint_to_bytes_outputi:nw
32012         { \__codepoint_to_bytes_auxii:Nnn C {#1} { 64 } }
32013         \__codepoint_to_bytes_outputii:nw
32014         { \__codepoint_to_bytes_auxiii:n {#1} }
32015     \else:
32016         \if_int_compare:w #1 < "10000 \exp_stop_f:
32017         \__codepoint_to_bytes_outputi:nw
32018         { \__codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 } }
32019         \__codepoint_to_bytes_outputii:nw
32020         {
32021             \__codepoint_to_bytes_auxiii:n
32022             { \int_div_truncate:nn {#1} { 64 } }
32023         }
32024         \__codepoint_to_bytes_outputiii:nw
32025         { \__codepoint_to_bytes_auxiii:n {#1} }
32026     \else:
32027         \__codepoint_to_bytes_outputi:nw
32028         {
32029             \__codepoint_to_bytes_auxii:Nnn F
32030             {#1} { 64 * 64 * 64 }
32031         }
32032         \__codepoint_to_bytes_outputii:nw
32033         {
32034             \__codepoint_to_bytes_auxiii:n
32035             { \int_div_truncate:nn {#1} { 64 * 64 } }
32036         }
32037         \__codepoint_to_bytes_outputiii:nw
32038         {
32039             \__codepoint_to_bytes_auxiii:n
32040             { \int_div_truncate:nn {#1} { 64 } }
32041         }
32042         \__codepoint_to_bytes_outputiv:nw
32043         { \__codepoint_to_bytes_auxiii:n {#1} }

```

```

32044         \fi:
32045     \fi:
32046 \else:
32047     \__codepoint_to_bytes_outputi:nw {#1}
32048 \fi:
32049 \__codepoint_to_bytes_end: { } { } { } { }
32050 }
32051 \cs_new:Npn \__codepoint_to_bytes_auxii:Nnn #1#2#3
32052 { "#10 + \int_div_truncate:nn {#2} {#3} }
32053 \cs_new:Npn \__codepoint_to_bytes_auxiii:n #1
32054 { \int_mod:nn {#1} { 64 } + 128 }
32055 \cs_new:Npn \__codepoint_to_bytes_outputi:nw
32056 #1 #2 \__codepoint_to_bytes_end: #3
32057 { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
32058 \cs_new:Npn \__codepoint_to_bytes_outputii:nw
32059 #1 #2 \__codepoint_to_bytes_end: #3#4
32060 { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
32061 \cs_new:Npn \__codepoint_to_bytes_outputiii:nw
32062 #1 #2 \__codepoint_to_bytes_end: #3#4#5
32063 {
32064     \__codepoint_to_bytes_output:fnn
32065     { \int_eval:n {#1} } { {#3} {#4} } {#2}
32066 }
32067 \cs_new:Npn \__codepoint_to_bytes_outputiv:nw
32068 #1 #2 \__codepoint_to_bytes_end: #3#4#5#6
32069 {
32070     \__codepoint_to_bytes_output:fnn
32071     { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
32072 }
32073 \cs_new:Npn \__codepoint_to_bytes_output:nnn #1#2#3
32074 {
32075     #3
32076     \__codepoint_to_bytes_end: #2 {#1}
32077 }
32078 \cs_generate_variant:Nn \__codepoint_to_bytes_output:nnn { f }
32079 \cs_new:Npn \__codepoint_to_bytes_end: { }

```

(End of definition for __kernel_codepoint_to_bytes:n and others.)

\codepoint_to_category:n Get the value and convert back to the string.

```

32080 \sys_if_engine luatex:TF
32081 {
32082     \cs_new:Npn \codepoint_to_category:n #1
32083     {
32084         \__kernel_codepoint_data:wn \int_eval:n { #1 } { category }
32085     }
32086 }
32087 {
32088     \cs_new:Npn \codepoint_to_category:n #1
32089     {
32090         \cs:w
32091         c__codepoint_category_
32092         \tex_romannumeral:D
32093         \__kernel_codepoint_data:nn { category } {#1}

```

```

32094         _str
32095     \cs_end:
32096 }
32097 }

```

(End of definition for `\codepoint_to_category:n`. This function is documented on page 302.)

```

\codepoint_to_nfd:n
__codepoint_to_nfd:n
__codepoint_to_nfd:nn
__codepoint_to_nfd:nnn
__codepoint_to_nfd:nnnn
32098 \cs_new:Npn \codepoint_to_nfd:n #1
32099 { \exp_args:Ne \__codepoint_to_nfd:n { \int_eval:n {#1} } }
32100 \cs_new:Npn \__codepoint_to_nfd:n #1
32101 { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
32102 \sys_if_engine_opentype:F
32103 {
32104     \cs_gset:Npn \__codepoint_to_nfd:n #1
32105     {
32106         \int_compare:nNnTF {#1} > { "80 }
32107         { \__codepoint_to_nfd:nn {#1} { 12 } }
32108         { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
32109     }
32110 }
32111 \cs_new:Npn \__codepoint_to_nfd:nn #1#2
32112 {
32113     \exp_args:Ne \__codepoint_to_nfd:nnn
32114     { \__codepoint_to_nfd:n {#1} } {#1} {#2}
32115 }
32116 \cs_new:Npn \__codepoint_to_nfd:nnn #1#2#3 { \__codepoint_to_nfd:nnnn #1 {#2} {#3} }
32117 \cs_new:Npn \__codepoint_to_nfd:nnnn #1#2#3#4
32118 {
32119     \int_compare:nNnTF {#1} = {#3}
32120     { \codepoint_generate:nn {#1} {#4} }
32121     {
32122         \__codepoint_to_nfd:nn {#1} {#4}
32123         \tl_if_blank:nF {#2}
32124         { \__codepoint_to_nfd:nn {#2} {#4} }
32125     }
32126 }

```

(End of definition for `\codepoint_to_nfd:n` and others. This function is documented on page 302.)

91.2 Data loader

Text operations requires data from the Unicode Consortium. Data read into Unicode engine formats is at best a small part of what we need, so there is a loader here to set up the appropriate data structures.

Where we need data for most or all of the Unicode range, we use the two-stage table approach recommended by the Unicode Consortium and demonstrated in a model implementation in Python in https://www.strchr.com/multi-stage_tables. This approach uses the `intarray` (fontdimen-based) data type as it is fast for random access and avoids significant hash table usage. In contrast, where only a small subset of codepoints are required, storage as macros is preferable. There is also some consideration of the effort needed to load data: see for example the grapheme breaking information,

which would be problematic to convert into a two-stage table but which can be used with reasonable performance in a small number of comma lists (at the cost that breaking at higher codepoint Hangul characters will be slightly slow).

`\c__codepoint_block_size_int`

Choosing the block size for the blocks in the two-stage approach is non-trivial: depending on the data stored, the optimal size for memory usage will vary. At the same time, for us there is also the question of load-time: larger blocks require longer comma lists as intermediates, so are slower. As this is going to be needed to use the data, we set it up outside of the group for clarity.

```
32127 \int_const:Nn \c__codepoint_block_size_int { 64 }
```

(End of definition for \c__codepoint_block_size_int.)

Parsing the data files can be the same way for all engines, but where they are stored as character tokens, the construction method depends on whether they are Unicode or 8-bit internally. Parsing is therefore done by common functions, with some data storage using engine-specific auxiliaries.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines.

LuaTeX does not use the parsing into multi stage tables. While it still stores the data in this format, this is handled by a faster Lua implementation in lua-uni-algos and reused here.

`\g__codepoint_data_ior`

```
32128 \ior_new:N \g__codepoint_data_ior
```

(End of definition for \g__codepoint_data_ior.)

We need some setup for the two-part table approach. The number of blocks we need will be variable, but the resulting size of the stage one table is predictable. For performance reasons, we therefore create the stage one tables now so they can be used immediately, and will later rename them as a constant tables. For each two-stage table construction, we need a comma list to hold the partial block and a couple of integers to track where we are up to. To avoid burning registers, the latter are stored in macros and are “fake” integers. We also avoid any **new** functions, keeping as much as possible local.

As we need both positive and negative values, case data requires one two-stage table for each transformation. In contrasts, general Unicode properties could be stored in one table with appropriate combination rules: that is not done at present but is likely to be added over time. Here, all that is needed is additional entries into the comma-list to create the structures.

Notice that in the standard expl3 way we are indexes position not offset: that does mean a little work later.

For integer values, everything is done using token lists to avoid losing registers or making global names.

```
32129 \sys_if_engine luatex:F
32130 {
32131   \group_begin:
32132   \clist_map_inline:nn
32133     { category , grapheme , lowercase , uppercase , wordbreak }
32134     {
32135       \cs_set_nopar:cpn { l__codepoint_ #1 _block_clist } { }
32136       \cs_set_nopar:cpn { l__codepoint_ #1 _block_tl } { 1 }
```

```

32137         \cs_set_nopar:cpn { l__codepoint_ #1 _pos_tl } { 0 }
32138         \cs_set_nopar:cpn { l__codepoint_ #1 _next_tl } { 0 }
32139         \intarray_new:cn { g__codepoint_ #1 _index_intarray }
32140         { \int_div_truncate:nn { "110000 } \c__codepoint_block_size_int }
32141     }

```

We also need to track the matched block: this is used dynamically so we only require one variable.

```

32142     \cs_set_nopar:Npn \l__codepoint_matched_block_tl { 0 }

```

For Unicode general category and the various breaking properties, there needs to be numerical representation of each possible value. As we need to go from string to number here, but the other way elsewhere, we set up fast mappings both ways, but one set local and the other as constants.

```

32143     \cs_set_nopar:Npn \l__codepoint_uppercase_default_tl { 0 }
32144     \cs_set_nopar:Npn \l__codepoint_lowercase_default_tl { 0 }
32145     \cs_set_protected:Npn \__codepoint_data_auxi:w #1#2#3
32146     {
32147         \quark_if_recursion_tail_stop:n {#3}
32148         \cs_set_nopar:cpn { l__codepoint_ #2 _ #3 _tl } {#1}
32149         \str_const:cn { c__codepoint_ #2 _ \tex_romannumeral:D #1 _str } {#3}
32150         \exp_args:Ne \__codepoint_data_auxi:w { \int_eval:n { #1 + 1 } } {#2}
32151     }
32152     \__codepoint_data_auxi:w { 1 } { category }
32153     { Lu } { Ll } { Lt } { Lm } { Lo }
32154     { Mn } { Me } { Mc }
32155     { Nd } { Nl } { No }
32156     { Zs } { Zl } { Zp }
32157     { Cc } { Cf } { Co } { Cs } { Cn }
32158     { Pd } { Ps } { Pe } { Pc } { Po } { Pi } { Pf }
32159     { Sm } { Sc } { Sk } { So }
32160     \q_recursion_tail
32161     \q_recursion_stop
32162     \cs_set_eq:NN \l__codepoint_category_default_tl \l__codepoint_category_Cn_tl
32163     \__codepoint_data_auxi:w { 1 } { grapheme }
32164     { Control }
32165     { CR } { LF } { ZWJ }
32166     { Extend }
32167     { L } { LV } { LVT } { T } { V }
32168     { Prepend }
32169     { Regional_Indicator }
32170     { SpacingMark }
32171     { Other }
32172     \q_recursion_tail
32173     \q_recursion_stop
32174     \cs_set_eq:NN \l__codepoint_grapheme_default_tl \l__codepoint_grapheme_Other_tl
32175     \__codepoint_data_auxi:w { 1 } { wordbreak }
32176     { Double_Quote } { Single_Quote }
32177     { CR } { LF } { Newline }
32178     { WSegSpace } { ZWJ }
32179     { Extend } { ExtendNumLet }
32180     { Regional_Indicator }
32181     { Format }
32182     { Katakana }
32183     { ALetter } { MidLetter } { Hebrew_Letter }

```

```

32184     { Numeric }{ MidNum } { MidNumLet }
32185     { Other }
32186     \q_recursion_tail
32187     \q_recursion_stop
32188     \cs_set_eq:NN \l__codepoint_wordbreak_default_tl \l__codepoint_wordbreak_Other_tl

```

For the the property fields, we need to use a two-step procedure as the file is ordered by class not codepoint. First, we parse the content into the hash table locally: there are around 1400 codepoints to handle, which is workable. Reading this is quite easy: we store any end-of-range codepoint and the class that applies. Conversion to a two-stage table is deferred to later.

```

32189     \cs_set_protected:Npn \__codepoint_data_auxi:w #1 ;~ #2 ~ #3 \q_stop
32190     { \__codepoint_data_auxii:w #1 .. \q_stop {#2} }
32191     \cs_set_protected:Npn \__codepoint_data_auxii:w #1 .. #2 \q_stop
32192     { \__codepoint_data_auxiii:w #1 ~ .. #2 ~ \q_stop }
32193     \cs_set_protected:Npn \__codepoint_data_auxiii:w #1 ~ #2 .. #3 ~ #4 \q_stop #5#6
32194     {
32195         \cs_set_nopar:cpe { l__codepoint_ #6 _ \tex_romannumeral:D "#1 _tl }
32196         {
32197             {#3}
32198             { \use:c { l__codepoint_ #6 _ #5 _tl } }
32199         }
32200     }
32201     \cs_set_protected:Npn \__codepoint_data_auxvi:w #1#2
32202     {
32203         \ior_open:Nn \g__codepoint_data_ior {#1}
32204         \ior_str_map_inline:Nn \g__codepoint_data_ior
32205         {
32206             \str_if_eq:eeF { \tl_head:w ##1 \c_hash_str \q_stop } { \c_hash_str }
32207             {
32208                 \tl_if_blank:nF {##1}
32209                 { \__codepoint_data_auxi:w ##1 \q_stop {#2} }
32210             }
32211         }
32212         \ior_close:N \g__codepoint_data_ior
32213     }
32214     \__codepoint_data_auxvi:w { GraphemeBreakProperty.txt } { grapheme }
32215     \__codepoint_data_auxvi:w { WordBreakProperty.txt } { wordbreak }

```

The set up for adding property data is now sorted.

```

32216     \cs_set_protected:Npn \__codepoint_data_property:nnnn #1#2#3#4
32217     {
32218         \int_compare:nNnT {#3} > { \use:c { l__codepoint_ #4 _next_tl } }
32219         {
32220             \__codepoint_range:nnv {#3} {#4}
32221             { l__codepoint_ #4 _default_tl }
32222         }
32223         \__codepoint_add:nn {#4} {#2}
32224         \tl_set:ce { l__codepoint_ #4 _next_tl } { \int_eval:n { #3 + 1 } }
32225         \tl_if_blank:nF {#1}
32226         {
32227             \__codepoint_range:nnn {"#1} {#4} {#2}
32228             \__codepoint_add:nn {#4} {#2}
32229             \tl_set:ce { l__codepoint_ #4 _next_tl } { \int_eval:n { "#1 + 1 } }
32230         }

```

```
32231     }
```

Parse the main Unicode data file and pull out the NFD and case changing data. The NFD data is stored on using the hash table approach and can yield a predictable number of codepoints: one or two. We also need the case data, which will be modified further below. To allow for finding ranges, the description of the codepoint needs to be carried forward.

```
32232     \cs_set_protected:Npn \__codepoint_data_auxi:w
32233     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
32234     {
32235         \tl_if_blank:nF {#6}
32236         {
32237             \tl_if_head_eq_charcode:nNF {#6} < % >
32238             { \__codepoint_data_auxii:w #1 ; #6 ~ \q_stop }
32239         }
32240         \__codepoint_data_auxiii:w #1 ; #2 ; #3 ;
32241     }
32242     \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ; #2 ~ #3 \q_stop
32243     {
32244         \tl_const:ce
32245         { c__codepoint_nfd_ \codepoint_str_generate:n {"#1} _tl }
32246         {
32247             {"#2}
32248             { \tl_if_blank:nF {#3} {"#3} }
32249         }
32250     }
```

The category data needs to be converted from a string to the numerical equivalent: a simple operation. The case data is going to be stored as an offset from the parent character, rather than an absolute value. We therefore deal with that plus the situation where a codepoint has no mapping data in one shot.

```
32251     \cs_set_protected:Npn \__codepoint_data_auxiii:w
32252     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ~ \q_stop
32253     {
32254         \use:e
32255         {
32256             \__codepoint_data_auxiv:w
32257             #1 ; #2 ;
32258             \__codepoint_data_category:n {#3} ;
32259             \__codepoint_data_offset:nn {#1} {#7} ;
32260             \__codepoint_data_offset:nn {#1} {#8} ;
32261             #9;
32262         }
32263     }
32264     \cs_set:Npn \__codepoint_data_category:n #1
32265     { \use:c { l__codepoint_category_ #1 _tl } }
32266     \cs_set:Npn \__codepoint_data_offset:nn #1#2
32267     {
32268         \tl_if_blank:nTF {#2}
32269         { 0 }
32270         { \int_eval:n { "#2 - "#1 } }
32271     }
```

To deal with ranges, we track the position of the next codepoint expected. If there is a gap, we deal with that separately: it could be a range or an unused part of the Unicode

space. As such, we deal with the current codepoint here whether or not there is range to fill in. Upper- and lowercase data go into the two-stage table, any titlecase exception is just stored in a macro. The data for the codepoint is added to the current block, and if that is now complete we move on to save the block. The case exceptions are all stored as codepoints, with a fixed number of balanced text as we know that there are never more than three.

```

32272 \cs_set_protected:Npn \__codepoint_data_auxiv:w #1 ; #2 ; #3 ; #4 ; #5 ; #6 ;
32273 {
32274   \int_compare:nNtT {"#1} > \l__codepoint_category_next_tl
32275   {
32276     \__codepoint_data_auxv:nnnw {#1} {#3} {#4} {#5}
32277     #2 Last> \q_stop
32278   }
32279   \__codepoint_add:nn { category } {#3}
32280   \__codepoint_add:nn { uppercase } {#4}
32281   \__codepoint_add:nn { lowercase } {#5}
32282   \int_compare:nNfT {#4} = { \__codepoint_data_offset:nn {#1} {#6} }
32283   {
32284     \tl_const:ce
32285     { c__codepoint_titlecase_ \codepoint_str_generate:n {"#1} _tl }
32286     { {"#6} { } { } { } }
32287   }
32288   \__codepoint_data_auxvi:nn { grapheme } {"#1}
32289   \__codepoint_data_auxvi:nn { wordbreak } {"#1}

```

To deal with the property data, we need to recover stored information and build the table. Generally we can assume that all codepoints in these files are in `UnicodeData.txt`, but there is one place a bit more work is needed. For Hangul syllables, the main file lists a range but within that we have different classifications for breaking: that needs to be handled with a second loop. The values for this range are hard-coded, skipping the end-of-range as that will be mopped up by the main loop.

```

32290   \int_compare:nNtT {"#1} = { "AC00 }
32291   {
32292     \int_step_inline:nnn { "AC01 } { "D7A2 }
32293     { \__codepoint_data_auxvi:nn { grapheme } {##1} }
32294   }
32295   \tl_set:Nx \l__codepoint_category_next_tl
32296   { \int_eval:n { "#1 + 1 } }
32297   \tl_set_eq:NN \l__codepoint_lowercase_next_tl \l__codepoint_category_next_tl
32298   \tl_set_eq:NN \l__codepoint_uppercase_next_tl \l__codepoint_category_next_tl
32299   }
32300   \cs_set_protected:Npn \__codepoint_add:nn #1#2
32301   {
32302     \clist_put_right:cn { l__codepoint_ #1 _block_clist } {#2}
32303     \int_compare:nNtT { \clist_count:c { l__codepoint_ #1 _block_clist } }
32304     = \c__codepoint_block_size_int
32305     { \__codepoint_save_blocks:nn {#1} { 1 } }
32306   }

```

Distinguish between a range and a gap, and pass on the appropriate value(s). The general category for unassigned characters is `Cn`, so we find the correct value once and then use that.

```

32307   \cs_set_protected:Npn \__codepoint_data_auxv:nnnw #1#2#3#4#5 Last> #6 \q_stop
32308   {

```

```

32309     \tl_if_blank:nTF {#6}
32310     {
32311         \__codepoint_range:nno {"#1} { category }
32312         \l__codepoint_category_default_tl
32313         \__codepoint_range:nnn {"#1} { uppercase } { 0 }
32314         \__codepoint_range:nnn {"#1} { lowercase } { 0 }
32315     }
32316     {
32317         \__codepoint_range:nnn {"#1} { category } {#2}
32318         \__codepoint_range:nnn {"#1} { uppercase } {#3}
32319         \__codepoint_range:nnn {"#1} { lowercase } {#4}
32320     }
32321 }

```

Recover and process grapheme data.

```

32322 \cs_set_protected:Npn \__codepoint_data_auxvi:nn #1#2
32323 {
32324     \cs_if_exist:cT
32325     { l__codepoint_ #1 _ \tex_romannumeral:D #2 _tl }
32326     {
32327         \exp_after:wN \exp_after:wN \exp_after:wN \__codepoint_data_property:nnnn
32328         \cs:w
32329         l__codepoint_ #1 _ \tex_romannumeral:D #2 _tl
32330         \cs_end: {#2} {#1}
32331     }
32332 }

```

Calculated the length of the range and the space remaining in the current block.

```

32333 \cs_set_protected:Npn \__codepoint_range:nnn #1#2
32334 {
32335     \exp_args:Nf \__codepoint_range_aux:nnn
32336     { \int_eval:n { #1 - \use:c { l__codepoint_ #2 _next_tl } } }
32337     {#2}
32338 }
32339 \cs_set_protected:Npn \__codepoint_range:nno { \exp_args:Nnno \__codepoint_range:nnn
32340 \cs_set_protected:Npn \__codepoint_range:nnv { \exp_args:Nnnv \__codepoint_range:nnn
32341 \cs_set_protected:Npn \__codepoint_range_aux:nnn #1#2
32342 {
32343     \exp_args:Nf \__codepoint_range:nnnn
32344     {
32345         \int_min:nn
32346         {#1}
32347         {
32348             \c__codepoint_block_size_int
32349             - \clist_count:c { l__codepoint_ #2 _block_clist }
32350         }
32351     }
32352     {#1} {#2}
32353 }

```

Here we want to do three things: add to and possibly complete the current block, add complete blocks quickly, then finish up the range in a final open block. We need to avoid as far as possible avoid dealing with every single codepoint, so the middle step is optimized.

```

32354 \cs_set_protected:Npn \__codepoint_range:nnnn #1#2#3#4

```

```

32355 {
32356   \prg_replicate:nn {#1}
32357   { \clist_put_right:cn { l__codepoint_ #3 _block_clist } {#4} }
32358 \int_compare:nNnT { \clist_count:c { l__codepoint_ #3 _block_clist } }
32359   = \c__codepoint_block_size_int
32360   { \__codepoint_save_blocks:nn {#3} { 1 } }
32361 \int_compare:nNnF
32362   { \int_div_truncate:nn { #2 - #1 } \c__codepoint_block_size_int } = 0
32363   {
32364     \tl_set:ce { l__codepoint_ #3 _block_clist }
32365     {
32366       \exp_args:NNe \use:nn \use_none:n
32367       { \prg_replicate:nn { \c__codepoint_block_size_int } { , #4 } }
32368     }
32369     \__codepoint_save_blocks:nn {#3}
32370     { \int_div_truncate:nn { (#2 - #1) } \c__codepoint_block_size_int }
32371   }
32372 \prg_replicate:nn
32373   { \int_mod:nn { #2 - #1 } \c__codepoint_block_size_int }
32374   { \clist_put_right:ce { l__codepoint_ #3 _block_clist } {#4} }
32375 }

```

To allow rapid comparison, each completed block is stored locally as a comma list: once all of the blocks have been created, they are converted into an `intarray` in one step. The aim here is to check the current block against those we've already used, and either match to an existing block or save a new block.

```

32376 \cs_set_protected:Npn \__codepoint_save_blocks:nn #1#2
32377 {
32378   \tl_set_eq:Nc \l__codepoint_matched_block_tl { l__codepoint_ #1 _block_tl }
32379   \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
32380   {
32381     \tl_if_eq:ccT { l__codepoint_ #1 _block_clist }
32382     { l__codepoint_ #1 _block_ ##1 _clist }
32383     { \tl_set:Nn \l__codepoint_matched_block_tl {##1} }
32384   }
32385   \int_compare:nNnT
32386     { \tl_use:c { l__codepoint_ #1 _block_tl } } = \l__codepoint_matched_block_tl
32387     {
32388       \clist_set_eq:cc
32389       {
32390         l__codepoint_ #1 _block_
32391         \tl_use:c { l__codepoint_ #1 _block_tl } _clist
32392       }
32393       { l__codepoint_ #1 _block_clist }
32394       \tl_set:ce { l__codepoint_ #1 _block_tl }
32395       { \int_eval:n { \tl_use:c { l__codepoint_ #1 _block_tl } + 1 } }
32396     }

```

Here, we avoid `\prg_replicate:nn` as the number of tokens generated would be high: that shows in the format dump (although $\text{\TeX}$ recovers memory during the subsequent runs).

```

32397 \int_step_inline:nnn
32398   { \tl_use:c { l__codepoint_ #1 _pos_tl } + 1 }
32399   { \tl_use:c { l__codepoint_ #1 _pos_tl } + #2 }

```

```

32400         {
32401             \exp_args:Nc \__kernel_intarray_gset:Nnn
32402             { g__codepoint_ #1 _index_intarray }
32403             {##1}
32404             \l__codepoint_matched_block_tl
32405         }
32406         \tl_set:ce { l__codepoint_ #1 _pos_tl }
32407         { \int_eval:n { \tl_use:c { l__codepoint_ #1 _pos_tl } + #2 } }
32408         \clist_clear:c { l__codepoint_ #1 _block_clist }
32409     }

```

Close out the final block, rename the first stage table, then combine all of the block comma-lists into one large second-stage table with offsets. As we use an index not an offset, there is a little back-and-forward to do.

```

32410     \cs_set_protected:Npn \__codepoint_finalize_blocks:n #1
32411     {
32412         \clist_map_inline:nn {#1}
32413         {
32414             \exp_args:Nnnv \__codepoint_range:nnn { "110000 } {##1}
32415             { l__codepoint_ ##1 _default_tl }
32416             \__codepoint_finalize_blocks_aux:n {##1}
32417         }
32418     }
32419     \cs_set_protected:Npn \__codepoint_finalize_blocks_aux:n #1
32420     {
32421         \cs_gset_eq:cc { c__codepoint_ #1 _index_intarray } { g__codepoint_ #1 _index_intarray }
32422         \cs_undefine:c { g__codepoint_ #1 _index_intarray }
32423         \intarray_new:cn { g__codepoint_ #1 _blocks_intarray }
32424         { ( \tl_use:c { l__codepoint_ #1 _block_tl } - 1 ) * \c__codepoint_block_size_int }
32425         \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
32426         {
32427             \exp_args:Nv \__codepoint_finalize_blocks:nnn
32428             { l__codepoint_ #1 _block_ ##1 _clist }
32429             {##1} {#1}
32430         }
32431         \cs_gset_eq:cc { c__codepoint_ #1 _blocks_intarray }
32432         { g__codepoint_ #1 _blocks_intarray }
32433         \cs_undefine:c { g__codepoint_ #1 _blocks_intarray }
32434     }
32435     \cs_set_protected:Npn \__codepoint_finalize_blocks:nnn #1#2#3
32436     {
32437         \exp_args:Nnf \__codepoint_finalize_blocks:nnnw { 1 }
32438         { \int_eval:n { ( #2 - 1 ) * \c__codepoint_block_size_int } }
32439         {#3}
32440         #1 , \q_recursion_tail , \q_recursion_stop
32441     }
32442     \cs_set_protected:Npn \__codepoint_finalize_blocks:nnnw #1#2#3#4 ,
32443     {
32444         \quark_if_recursion_tail_stop:n {#4}
32445         \intarray_gset:cnn { g__codepoint_ #3 _blocks_intarray }
32446         { #1 + #2 }
32447         {#4}
32448         \exp_args:Nf \__codepoint_finalize_blocks:nnnw
32449         { \int_eval:n { #1 + 1 } } {#2} {#3}

```

```
32450 }
```

With the setup done, read the main data file: it's easiest to do that as a token list with spaces retained.

```
32451 \ior_open:Nn \g__codepoint_data_ior { UnicodeData.txt }
32452 \char_set_catcode_space:n { '\ }%
32453 \ior_map_variable:NNn \g__codepoint_data_ior \l__codepoint_tmpa_tl
32454 {%
32455 \if_meaning:w \l__codepoint_tmpa_tl \c_space_tl
32456 \exp_after:wN \ior_map_break:
32457 \fi:
32458 \exp_after:wN \__codepoint_data_auxi:w \l__codepoint_tmpa_tl \q_stop
32459 }%
32460 \char_set_catcode_ignore:n { '\ }%
```

Blocks need to be finalized once they have been read as this relies on knowing the last real codepoint. So the tables for `UnicodeData.txt` are closed out before reading additional files.

```
32461 \__codepoint_finalize_blocks:n
32462 { category , grapheme , lowercase , uppercase , wordbreak }
32463 \group_end:
```

```
\__kernel_codepoint_data:nn
\__codepoint_data:nnn
```

Recover data from a two-stage table: entirely generic as this applies to all tables (as we use the same block size for all of them). Notice that as we use indices not offsets we have to shuffle out-by-one issues. This function is needed *before* loading the special casing data, as there we need to be able to check the standard case mappings.

This macro is not defined in LuaTeX where the data is provided in a more pre-processed form in `\__kernel_codepoint_data:wn`.

```
32464 \cs_new:Npn \__kernel_codepoint_data:nn #1#2
32465 {
32466 \exp_args:Nf \__codepoint_data:nnn
32467 {
32468 \int_eval:n
32469 {
32470 \c__codepoint_block_size_int *
32471 (
32472 \intarray_item:cn { c__codepoint_ #1 _index_intarray }
32473 {
32474 \int_div_truncate:nn {#2}
32475 \c__codepoint_block_size_int
32476 + 1
32477 }
32478 - 1
32479 )
32480 }
32481 }
32482 {#2} {#1}
32483 }
32484 \cs_new:Npn \__codepoint_data:nnn #1#2#3
32485 {
32486 \intarray_item:cn { c__codepoint_ #3 _blocks_intarray }
32487 { #1 + \int_mod:nn {#2} \c__codepoint_block_size_int + 1 }
32488 }
32489 }
```

(End of definition for `\__kernel_codepoint_data:nn` and `\__codepoint_data:nnn`.)

The remaining files use the $\TeX$ based parser also in Lua $\TeX$.

The other data files all use C-style comments so we have to worry about `#` tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

32490 \group_begin:
32491   \ior_open:Nn \g__codepoint_data_ior { CaseFolding.txt }
32492   \cs_set_protected:Npn \__codepoint_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
32493   {
32494     \if:w \tl_head:n { #2 ? } C
32495     \reverse_if:N \if_int_compare:w
32496       \__codepoint_lowercase_base_mapping:n { "#1 }
32497       = "#3 ~
32498       \tl_const:ce
32499       { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
32500       { {"#3} { } { } }
32501     \fi:
32502     \else:
32503       \if:w \tl_head:n { #2 ? } F
32504       \__codepoint_data_auxii:w #1 ~ #3 ~ \q_stop
32505     \fi:
32506     \fi:
32507   }
32508   \sys_if_engine luatex:TF
32509   {
32510     \cs_set:Npn \__codepoint_lowercase_base_mapping:n #1
32511     {
32512       \__kernel_codepoint_data:wn #1 { lowercase }
32513     }
32514   }
32515   {
32516     \cs_set:Npn \__codepoint_lowercase_base_mapping:n #1
32517     {
32518       \int_eval:n { \__kernel_codepoint_data:nn { lowercase } {#1} + #1 }
32519     }
32520   }

```

Here, `#4` can have a trailing space, so we tidy up a bit at the cost of speed for these small number of cases it applies to.

```

32521   \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
32522   {
32523     \tl_const:ce { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
32524     {
32525       {"#2}
32526       {"#3}
32527       { \tl_if_blank:nF {#4} { " \int_to_Hex:n {"#4} } }
32528     }
32529   }
32530   \ior_str_map_inline:Nn \g__codepoint_data_ior
32531   {
32532     \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
32533     \__codepoint_data_auxi:w #1 \q_stop

```

```

32534     \fi:
32535   }
32536   \ior_close:N \g__codepoint_data_ior

For upper- and lowercasing special situations, there is a bit more to do as we also have
titlecasing to consider, plus we need to stop part-way through the file.

32537   \ior_open:Nn \g__codepoint_data_ior { SpecialCasing.txt }
32538   \cs_set_protected:Npn \__codepoint_data_auxi:w
32539     #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
32540   {
32541     \use:n { \__codepoint_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
32542     \use:n { \__codepoint_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
32543     \str_if_eq:nnF {#3} {#4}
32544       { \use:n { \__codepoint_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
32545   }
32546   \cs_set_protected:Npn \__codepoint_data_auxii:w
32547     #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
32548   {
32549     \tl_if_empty:nF {#4}
32550     {
32551       \tl_const:ce { c__codepoint_ #2 case_ \codepoint_str_generate:n {"#1} _tl }
32552       {
32553         {"#3}
32554         {"#4}
32555         { \tl_if_blank:nF {#5} {"#5} }
32556       }
32557     }
32558   }
32559   \ior_str_map_inline:Nn \g__codepoint_data_ior
32560   {
32561     \str_if_eq:eeTF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
32562     {
32563       \str_if_eq:eeT
32564         {#1}
32565         { \c_hash_str \c_space_tl Conditional-Mappings }
32566         { \ior_map_break: }
32567     }
32568     { \__codepoint_data_auxi:w #1 \q_stop }
32569   }
32570   \ior_close:N \g__codepoint_data_ior
32571 \group_end:

```

`\__kernel_codepoint_case:nn` With the core data files loaded, there is now a need to provide access to this information for other modules. That is done here such that case folding can also be covered. At this level, all that needs to be returned is the

```

\__codepoint_case:nnn
\__codepoint_uppercase:n
\__codepoint_lowercase:n
\__codepoint_titlecase:n
\__codepoint_casefold:n
\__codepoint_case:nn
32572 \cs_new:Npn \__kernel_codepoint_case:nn #1#2
32573 {
32574   \exp_args:Ne \__codepoint_case:nnn
32575   { \codepoint_str_generate:n {#2} } {#1} {#2}
32576 }
32577 \cs_new:Npn \__codepoint_case:nnn #1#2#3
32578 {
32579   \cs_if_exist:cTF { c__codepoint_ #2 _ #1 _tl }
32580   {

```

```

32581         \tl_use:c
32582         { c__codepoint_ #2 _ #1 _tl }
32583     }
32584     { \use:c { __codepoint_ #2 :n } {#3} }
32585 }
32586 \cs_new:Npn \__codepoint_uppercase:n { \__codepoint_case:nn { uppercase } }
32587 \cs_new:Npn \__codepoint_lowercase:n { \__codepoint_case:nn { lowercase } }
32588 \sys_if_engine luatex:TF
32589 {
32590     \cs_new:Npn \__codepoint_titlecase:n { \__codepoint_case:nn { titlecase } }
32591 }
32592 {
32593     \cs_new:Npn \__codepoint_titlecase:n { \__codepoint_case:nn { uppercase } }
32594 }
32595 \cs_new:Npn \__codepoint_casefold:n { \__codepoint_case:nn { lowercase } }
32596 \sys_if_engine luatex:TF
32597 {
32598     \cs_new:Npn \__codepoint_case:nn #1#2
32599     {
32600         { \__kernel_codepoint_data:wn \int_eval:n {#2} {#1} }
32601         { }
32602         { }
32603     }
32604 }
32605 {
32606     \cs_new:Npn \__codepoint_case:nn #1#2
32607     {
32608         { \int_eval:n { \__kernel_codepoint_data:nn {#1} {#2} + #2 } }
32609         { }
32610         { }
32611     }
32612 }

```

(End of definition for __kernel_codepoint_case:nn and others.)

__kernel_codepoint_to_grapheme_class:n Get the value and convert back to the string: not currently public but otherwise very similar to \codepoint_to_category:n.

__kernel_codepoint_to_wordbreak_class:n

```

32613 \sys_if_engine luatex:TF
32614 {
32615     \cs_new:Npn \__kernel_codepoint_to_grapheme_class:n #1
32616     {
32617         \__kernel_codepoint_data:wn \int_eval:n {#1} { grapheme }
32618     }
32619     \cs_new:Npn \__kernel_codepoint_to_wordbreak_class:n #1
32620     {
32621         \__kernel_codepoint_data:wn \int_eval:n {#1} { wordbreak }
32622     }
32623 }
32624 {
32625     \cs_new:Npn \__kernel_codepoint_to_grapheme_class:n #1
32626     {
32627         \cs:w
32628         c__codepoint_grapheme_
32629         \tex_romannumeral:D

```



```

32630         \__kernel_codepoint_data:nn { grapheme } {#1}
32631         _str
32632     \cs_end:
32633 }
32634 \cs_new:Npn \__kernel_codepoint_to_wordbreak_class:n #1
32635 {
32636     \cs:w
32637     c__codepoint_wordbreak_
32638     \tex_romannumeral:D
32639     \__kernel_codepoint_data:nn { wordbreak } {#1}
32640     _str
32641     \cs_end:
32642 }
32643 }

```

(End of definition for __kernel_codepoint_to_grapheme_class:n and __kernel_codepoint_to_wordbreak_class:n.)

__codepoint_nfd:n A simple interface.

```

\__codepoint_nfd:nn
32644 \sys_if_engine luatex:TF
32645 {
32646     \cs_new:Npn \__codepoint_nfd:n #1
32647     { \__codepoint_nfd:w \__int_eval:w #1 \__int_eval_end: }
32648 }
32649 {
32650     \cs_new:Npn \__codepoint_nfd:n #1
32651     { \exp_args:Ne \__codepoint_nfd:nn { \codepoint_str_generate:n {#1} } {#1} }
32652     \cs_new:Npn \__codepoint_nfd:nn #1#2
32653     {
32654         \tl_if_exist:cTF { c__codepoint_nfd_ #1 _tl }
32655         { \tl_use:c { c__codepoint_nfd_ #1 _tl } }
32656         { {#2} { } }
32657     }
32658 }

```

(End of definition for __codepoint_nfd:n and __codepoint_nfd:nn.)

```

32659 </code>

```

Chapter 92

l3text implementation

```
32660 <*code>
32661 <@@=text>
32662 \cs_generate_variant:Nn \tl_if_head_eq_meaning_p:nN { o }
```

92.1 Internal auxiliaries

`\s__text_stop` Internal scan marks.

```
32663 \scan_new:N \s__text_stop
```

(End of definition for \s__text_stop.)

`\q__text_nil` Internal quarks.

```
32664 \quark_new:N \q__text_nil
```

(End of definition for \q__text_nil.)

`\__text_quark_if_nil_p:n` Branching quark conditional.

```
\__text_quark_if_nil:nTF
32665 \__kernel_quark_new_conditional:Nn \__text_quark_if_nil:n { TF }
```

(End of definition for __text_quark_if_nil:nTF.)

`\q__text_recursion_tail` Internal recursion quarks.

```
\q__text_recursion_stop
32666 \quark_new:N \q__text_recursion_tail
32667 \quark_new:N \q__text_recursion_stop
```

(End of definition for \q__text_recursion_tail and \q__text_recursion_stop.)

`\__text_use_i_delimit_by_q_recursion_stop:nw` Functions to gobble up to a quark.

```
32668 \cs_new:Npn \__text_use_i_delimit_by_q_recursion_stop:nw
32669 #1 #2 \q__text_recursion_stop {#1}
```

(End of definition for __text_use_i_delimit_by_q_recursion_stop:nw.)

`\__text_if_q_recursion_tail_stop_do:Nn` Functions to query recursion quarks.

```
\__text_if_q_recursion_tail_stop_do:nn
32670 \__kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:Nn
32671 \__kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:nn
```

(End of definition for `\__text_if_q_recursion_tail_stop_do:Nn` and `\__text_if_q_recursion_tail_stop_do:nn`.)

`\s__text_recursion_tail` Internal scan marks quarks.

`\s__text_recursion_stop` 32672 `\scan_new:N \s__text_recursion_tail`
32673 `\scan_new:N \s__text_recursion_stop`

(End of definition for `\s__text_recursion_tail` and `\s__text_recursion_stop`.)

`\__text_use_i_delimit_by_s_recursion_stop:nw` Functions to gobble up to a scan mark.

32674 `\cs_new:Npn \__text_use_i_delimit_by_s_recursion_stop:nw`
32675 `#1 #2 \s__text_recursion_stop {#1}`

(End of definition for `\__text_use_i_delimit_by_s_recursion_stop:nw`.)

`\__text_if_s_recursion_tail_stop_do:Nn` Functions to query recursion scan marks. Slower than a quark test but needed to avoid issues in the outer expansion loop with unterminated `\romannumeral` primitives.

32676 `\cs_new:Npn \__text_if_s_recursion_tail_stop_do:Nn #1`
32677 `{`
32678 `\bool_lazy_and:nnTF`
32679 `{ \cs_if_eq_p:NN \s__text_recursion_tail #1 }`
32680 `{ \str_if_eq_p:nn { \s__text_recursion_tail } {#1} }`
32681 `{ \__text_use_i_delimit_by_s_recursion_stop:nw }`
32682 `{ \use_none:n }`
32683 `}`

(End of definition for `\__text_if_s_recursion_tail_stop_do:Nn`.)

92.2 Utilities

`\__text_sep:`

32684 `\cs_new_eq:NN \__text_sep: \__kernel_int_sep:`

(End of definition for `\__text_sep:.`)

`\__text_token_to_explicit:N`

`\__text_token_to_explicit_char:N`

`\__text_token_to_explicit_cs:N`

`\__text_token_to_explicit_cs_aux:N`

`\__text_token_to_explicit:n`

`\__text_token_to_explicit_auxi:w`

`\__text_token_to_explicit_auxii:w`

`\__text_token_to_explicit_auxiii:w`

The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

32685 `\group_begin:`
32686 `\char_set_catcode_active:n { 0 }`
32687 `\cs_new:Npn \__text_token_to_explicit:N #1`
32688 `{`
32689 `\if_catcode:w \exp_not:N #1`
32690 `\if_catcode:w \scan_stop: \exp_not:N #1`
32691 `\scan_stop:`
32692 `\else:`
32693 `\exp_not:N ^^@`
32694 `\fi:`
32695 `\exp_after:wN \__text_token_to_explicit_cs:N`
32696 `\else:`
32697 `\exp_after:wN \__text_token_to_explicit_char:N`
32698 `\fi:`
32699 `#1`
32700 `}`
32701 `\group_end:`

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

32702 \cs_new:Npn \__text_token_to_explicit_cs:N #1
32703 {
32704   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
32705   \exp_after:wN \use:nn \exp_after:wN
32706     \__text_token_to_explicit_cs_aux:N
32707   \else:
32708     \exp_after:wN \exp_not:n
32709   \fi:
32710   {#1}
32711 }
32712 \cs_new:Npn \__text_token_to_explicit_cs_aux:N #1
32713 {
32714   \bool_lazy_or:nnTF
32715     { \token_if_chardef_p:N #1 }
32716     { \token_if_mathchardef_p:N #1 }
32717     {
32718       \char_generate:nn {#1}
32719       {
32720         \if_int_compare:w \char_value_catcode:n {#1} = 10 \exp_stop_f:
32721         10
32722         \else:
32723         12
32724         \fi:
32725       }
32726     }
32727     {#1}
32728 }

```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the $\text{T}_{\text{E}}\text{X}$ messages are either the *something* character *char* or the *type* *char*.

```

32729 \cs_new:Npn \__text_token_to_explicit_char:N #1
32730 {
32731   \if:w
32732     \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
32733     \token_to_str:N #1 #1
32734   \else:
32735     AB
32736   \fi:
32737   \exp_after:wN \exp_not:n
32738   \else:
32739     \exp_after:wN \__text_token_to_explicit:n
32740   \fi:
32741   {#1}
32742 }
32743 \cs_new:Npn \__text_token_to_explicit:n #1
32744 {
32745   \exp_after:wN \__text_token_to_explicit_auxi:w
32746   \int_value:w
32747   \if_catcode:w \c_group_begin_token #1 1 \else:

```

```

32748 \if_catcode:w \c_group_end_token #1 2 \else:
32749 \if_catcode:w \c_math_toggle_token #1 3 \else:
32750 \if_catcode:w ## #1 6 \else:
32751 \if_catcode:w ^ #1 7 \else:
32752 \if_catcode:w \c_math_subscript_token #1 8 \else:
32753 \if_catcode:w \c_space_token #1 10 \else:
32754 \if_catcode:w A #1 11 \else:
32755 \if_catcode:w + #1 12 \else:
32756 4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
32757 \exp_after:wN \_text_sep:
32758 \token_to_meaning:N #1 \s__text_stop
32759 }
32760 \cs_new:Npn \_text_token_to_explicit_auxi:w #1 \_text_sep: #2 \s__text_stop
32761 {
32762 \char_generate:nn
32763 {
32764 \if_int_compare:w #1 < 9 \exp_stop_f:
32765 \exp_after:wN \_text_token_to_explicit_auxii:w
32766 \else:
32767 \exp_after:wN \_text_token_to_explicit_auxiii:w
32768 \fi:
32769 #2
32770 }
32771 {#1}
32772 }
32773 \exp_last_unbraced:NNNNo \cs_new:Npn \_text_token_to_explicit_auxii:w
32774 #1 { \tl_to_str:n { character ~ } } { ' }
32775 \cs_new:Npn \_text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

```

(End of definition for `\_text_token_to_explicit:N` and others.)

`\_text_char_catcode:N` An idea from `l3char`: we need to get the category code of a specific token, not the general case.

```

32776 \cs_new:Npn \_text_char_catcode:N #1
32777 {
32778 \if_catcode:w \exp_not:N #1 \c_math_toggle_token
32779 3
32780 \else:
32781 \if_catcode:w \exp_not:N #1 \c_alignment_token
32782 4
32783 \else:
32784 \if_catcode:w \exp_not:N #1 \c_math_superscript_token
32785 7
32786 \else:
32787 \if_catcode:w \exp_not:N #1 \c_math_subscript_token
32788 8
32789 \else:
32790 \if_catcode:w \exp_not:N #1 \c_space_token
32791 10
32792 \else:
32793 \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
32794 11
32795 \else:
32796 \if_catcode:w \exp_not:N #1 \c_catcode_other_token

```

```

32797             12
32798             \else:
32799             13
32800             \fi:
32801         \fi:
32802     \fi:
32803 \fi:
32804 \fi:
32805 \fi:
32806 \fi:
32807 }

```

(End of definition for `\_text\_char\_catcode:N`.)

`\_text\_if\_expandable:NTF` Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

32808 \prg_new_conditional:Npnn \_text\_if\_expandable:N #1 { T , F , TF }
32809 {
32810     \token\_if\_expandable:NTF #1
32811     {
32812         \bool_lazy_any:nTF
32813         {
32814             { \token\_if\_protected\_macro\_p:N      #1 }
32815             { \token\_if\_protected\_long\_macro\_p:N #1 }
32816             { \token\_if\_eq\_meaning\_p:NN \q\_text\_recursion\_tail #1 }
32817         }
32818         { \prg_return_false: }
32819         { \prg_return_true: }
32820     }
32821     { \prg_return_false: }
32822 }

```

(End of definition for `\_text\_if\_expandable:NTF`.)

92.3 Codepoint utilities

For working with codepoints in an engine-neutral way.

`\_text\_codepoint\_process:nN` Grab a codepoint and apply some code to it: here #1 should expect one following *balanced text*.

```

\_text\_codepoint\_process\_aux:nN
\_text\_codepoint\_process:nNN
\_text\_codepoint\_process:nNNN
\_text\_codepoint\_process:nNNNN
32823 \sys\_if\_engine\_opentype:TF
32824 {
32825     \cs\_new:Npn \_text\_codepoint\_process:nN #1#2 { #1 {#2} }
32826 }
32827 {
32828     \cs\_new:Npe \_text\_codepoint\_process:nN #1#2
32829     {
32830         \exp\_not:N \int\_compare:nNnTF {'#2} > { "80 }
32831         {
32832             \sys\_if\_engine\_pdftex:TF
32833             { \exp\_not:N \_text\_codepoint\_process\_aux:nN }
32834             {
32835                 \exp\_not:N \int\_compare:nNnTF {'#2} > { "FF }

```

```

32836             { \exp_not:N \use:n }
32837             { \exp_not:N \__text_codepoint_process_aux:nN }
32838         }
32839     }
32840     { \exp_not:N \use:n }
32841     {#1} #2
32842 }
32843 \cs_new:Npn \__text_codepoint_process_aux:nN #1#2
32844 {
32845     \int_compare:nNnTF { '#2 } < { "EO }
32846     { \__text_codepoint_process:nNN }
32847     {
32848         \int_compare:nNnTF { '#2 } < { "FO }
32849         { \__text_codepoint_process:nNNN }
32850         { \__text_codepoint_process:nNNNN }
32851     }
32852     {#1} #2
32853 }
32854 \cs_new:Npn \__text_codepoint_process:nNN #1#2#3
32855 { #1 {#2#3} }
32856 \cs_new:Npn \__text_codepoint_process:nNNN #1#2#3#4
32857 { #1 {#2#3#4} }
32858 \cs_new:Npn \__text_codepoint_process:nNNNN #1#2#3#4#5
32859 { #1 {#2#3#4#5} }
32860 }

```

(End of definition for __text_codepoint_process:nN and others.)

<pre> __text_codepoint_compare_p:nNn __text_codepoint_compare:nNnTF __text_codepoint_from_chars:Nw __text_codepoint_from_chars_aux:Nw __text_codepoint_from_chars:N __text_codepoint_from_chars:NN __text_codepoint_from_chars:NNN __text_codepoint_from_chars:NNNN </pre>	Allows comparison for all engines using a first “character” followed by a codepoint. <pre> 32861 \sys_if_engine_opentype:TF 32862 { 32863 \prg_new_conditional:Npnn 32864 __text_codepoint_compare:nNn #1#2#3 { TF , p } 32865 { 32866 \int_compare:nNnTF { '#1 } #2 { #3 } 32867 \prg_return_true: \prg_return_false: 32868 } 32869 \cs_new:Npn __text_codepoint_from_chars:Nw #1 { '#1 } 32870 } 32871 { 32872 \prg_new_conditional:Npnn 32873 __text_codepoint_compare:nNn #1#2#3 { TF , p } 32874 { 32875 \int_compare:nNnTF { __text_codepoint_from_chars:Nw #1 } 32876 #2 { #3 } 32877 \prg_return_true: \prg_return_false: 32878 } 32879 \cs_new:Npe __text_codepoint_from_chars:Nw #1 32880 { 32881 \exp_not:N \if_int_compare:w '#1 > "80 \exp_not:N \exp_stop_f: 32882 \sys_if_engine_pdftex:TF 32883 { 32884 \exp_not:N \exp_after:wN 32885 \exp_not:N __text_codepoint_from_chars_aux:Nw </pre>

```

32886     }
32887     {
32888         \exp_not:N \if_int_compare:w '#1 > "FF \exp_not:N \exp_stop_f:
32889         \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
32890         \exp_not:N \exp_after:wN
32891         \exp_not:N \__text_codepoint_from_chars:N
32892         \exp_not:N \else:
32893         \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
32894         \exp_not:N \exp_after:wN
32895         \exp_not:N \__text_codepoint_from_chars_aux:Nw
32896         \exp_not:N \fi:
32897     }
32898     \exp_not:N \else:
32899     \exp_not:N \exp_after:wN \exp_not:N \__text_codepoint_from_chars:N
32900     \exp_not:N \fi:
32901     #1
32902 }
32903 \cs_new:Npn \__text_codepoint_from_chars_aux:Nw #1
32904 {
32905     \if_int_compare:w '#1 < "E0 \exp_stop_f:
32906     \exp_after:wN \__text_codepoint_from_chars:NN
32907     \else:
32908     \if_int_compare:w '#1 < "F0 \exp_stop_f:
32909     \exp_after:wN \exp_after:wN \exp_after:wN
32910     \__text_codepoint_from_chars:NNN
32911     \else:
32912     \exp_after:wN \exp_after:wN \exp_after:wN
32913     \__text_codepoint_from_chars:NNNN
32914     \fi:
32915     \fi:
32916     #1
32917 }
32918 \cs_new:Npn \__text_codepoint_from_chars:N #1 {'#1}
32919 \cs_new:Npn \__text_codepoint_from_chars:NN #1#2
32920 { ('#1 - "C0) * "40 + '#2 - "80 }
32921 \cs_new:Npn \__text_codepoint_from_chars:NNN #1#2#3
32922 { ('#1 - "E0) * "1000 + ('#2 - "80) * "40 + '#3 - "80 }
32923 \cs_new:Npn \__text_codepoint_from_chars:NNNN #1#2#3#4
32924 {
32925     ('#1 - "F0) * "40000
32926     + ('#2 - "80) * "1000
32927     + ('#3 - "80) * "40
32928     + '#4 - "80
32929 }
32930 }

```

(End of definition for __text_codepoint_compare:nNnTF and others.)

92.4 Configuration variables

\l_text_accents_tl Used to be used for excluding these ideas from expansion: now deprecated.

\l_text_letterlike_tl 32931 \tl_new:N \l_text_accents_tl

32932 \tl_new:N \l_text_letterlike_tl

(End of definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`.)

`\l_text_case_exclude_arg_tl` Non-text arguments, including covering the case of `\protected@edef` applied to `\cite`.

```

32933 \tl_new:N \l_text_case_exclude_arg_tl
32934 \tl_set:N \l_text_case_exclude_arg_tl
32935 {
32936   \exp_not:n { \begin \cite \end \label \ref }
32937   \exp_not:c { cite ~ }
32938   \exp_not:n { \babelshorthand }
32939 }

```

(End of definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 306.)

`\l_text_math_arg_tl` Math mode as arguments.

```

32940 \tl_new:N \l_text_math_arg_tl
32941 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }

```

(End of definition for `\l_text_math_arg_tl`. This variable is documented on page 306.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```

32942 \tl_new:N \l_text_math_delims_tl
32943 \tl_set:Nn \l_text_math_delims_tl { $ $ \ ( \ ) }

```

(End of definition for `\l_text_math_delims_tl`. This variable is documented on page 306.)

`\l_text_expand_exclude_tl` Commands which need not to expand. We start with a somewhat historical list, and tidy up if possible.

```

32944 \tl_new:N \l_text_expand_exclude_tl
32945 \tl_set:Nn \l_text_expand_exclude_tl
32946 { \begin \cite \end \label \ref }
32947 \bool_lazy_and:nnT
32948 { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
32949 { \tl_if_exist_p:N \@expl@finalise@setup@@ }
32950 {
32951   \tl_gput_right:Nn \@expl@finalise@setup@@
32952   {
32953     \tl_gput_right:Nn \@kernel@after@begindocument
32954     {
32955       \group_begin:
32956       \cs_set_protected:Npn \__text_tmp:w #1
32957       {
32958         \tl_clear:N \l_text_expand_exclude_tl
32959         \tl_map_inline:nn {#1}
32960         {
32961           \bool_lazy_any:nF
32962           {
32963             { \token_if_protected_macro_p:N ##1 }
32964             { \token_if_protected_long_macro_p:N ##1 }
32965             {
32966               \str_if_eq_p:ee
32967               { \cs_replacement_spec:N ##1 }
32968               { \exp_not:n { \protect ##1 } \c_space_tl }
32969             }
32970           }
32971         }
32972       }
32973     }
32974   }

```

```

32971             { \tl_put_right:Nn \l_text_expand_exclude_tl {##1} }
32972         }
32973     }
32974     \exp_args:NV \__text_tmp:w \l_text_expand_exclude_tl
32975     \exp_args:NNNV \group_end:
32976     \tl_set:Nn \l_text_expand_exclude_tl \l_text_expand_exclude_tl
32977 }
32978 }
32979 }

```

(End of definition for `\l_text_expand_exclude_tl`. This variable is documented on page [306](#).)

`\l__text_math_mode_tl` Used to control math mode output: internal as there is a dedicated setter.

```

32980 \tl_new:N \l__text_math_mode_tl

```

(End of definition for `\l__text_math_mode_tl`.)

92.5 Expansion to formatted text

`\c__text_chardef_space_token` Markers for implicit char handling.

```

32981 \tex_global:D \tex_chardef:D \c__text_chardef_space_token = '\ %
32982 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_space_token = '\ %
32983 \tex_global:D \tex_chardef:D \c__text_chardef_group_begin_token = '\{ % '\}
32984 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_begin_token = '\{ % '\} '\{
32985 \tex_global:D \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\{
32986 \tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %

```

(End of definition for `\c__text_chardef_space_token` and others.)

`\text_expand:n` After precautions against `&` tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.) The outer loop has to use scan marks as delimiters to protect against unterminated `\romannumeral` usage in the input.

```

32987 \cs_new:Npn \text_expand:n #1
32988 {
32989     \__kernel_exp_not:w \exp_after:wN
32990     {
32991         \exp:w
32992         \__text_expand:n {#1}
32993     }
32994 }
32995 \cs_new:Npn \__text_expand:n #1
32996 {
32997     \group_align_safe_begin:
32998     \__text_expand_loop:w #1
32999     \s__text_recursion_tail \s__text_recursion_stop
33000     \__text_expand_result:n { }
33001 }

```

`\__text_expand_result:n`
`\__text_expand_store:n`
`\__text_expand_store:o`
`\__text_expand_store:nw`
`\__text_expand_end:w`
`\__text_expand_loop:w`
`\__text_expand_group:n`
`\__text_expand_space:w`
`\__text_expand_N_type:N`
`\__text_expand_math_search:NNN`
`\__text_expand_math_loop:Nw`
`\__text_expand_math_N_type:NN`
`\__text_expand_math_group:Nn`
`\__text_expand_math_space:Nw`
`\__text_expand_explicit:N`
`\__text_expand_exclude:N`
`\__text_expand_exclude_switch:Nnnnn`
`\__text_expand_exclude:nN`
`\__text_expand_exclude:NN`
`\__text_expand_exclude:Nw`
`\__text_expand_exclude:Nnn`
`\__text_expand_accent:N`
`\__text_expand_accent:NN`
`\__text_expand_letterlike:N`
`\__text_expand_letterlike:NN`
`\__text_expand_cs:N`
`\__text_expand_protect:w`
`\__text_expand_protect:N`
`\__text_expand_protect:nN`
`\__text_expand_protect:Nw`
`\__text_expand_testopt:N`

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

```

33002 \cs_new:Npn \__text_expand_store:n #1
33003 { \__text_expand_store:nw {#1} }
33004 \cs_generate_variant:Nn \__text_expand_store:n { o }
33005 \cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
33006 { #2 \__text_expand_result:n { #3 #1 } }
33007 \cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
33008 {
33009   \group_align_safe_end:
33010   \exp_end:
33011   #2
33012 }

```

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

33013 \cs_new:Npn \__text_expand_loop:w #1 \s__text_recursion_stop
33014 {
33015   \tl_if_head_is_N_type:nTF {#1}
33016   { \__text_expand_N_type:N }
33017   {
33018     \tl_if_head_is_group:nTF {#1}
33019     { \__text_expand_group:n }
33020     { \__text_expand_space:w }
33021   }
33022   #1 \s__text_recursion_stop
33023 }
33024 \cs_new:Npn \__text_expand_group:n #1
33025 {
33026   \__text_expand_store:o
33027   {
33028     \exp_after:wN
33029     {
33030       \exp:w
33031       \__text_expand:n {#1}
33032     }
33033   }
33034   \__text_expand_loop:w
33035 }
33036 \exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
33037 {
33038   \__text_expand_store:n { ~ }
33039   \__text_expand_loop:w
33040 }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need `\quark_if_recursion_tail_stop:N` first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

33041 \cs_new:Npn \__text_expand_N_type:N #1
33042 {
33043   \__text_if_s_recursion_tail_stop_do:Nn #1

```

```

33044     { \_text_expand_end:w }
33045 \exp_after:wN \_text_expand_math_search:NNN
33046 \exp_after:wN #1 \l_text_math_delims_tl
33047 \q_text_recursion_tail \q_text_recursion_tail
33048 \q_text_recursion_stop
33049 }
33050 \cs_new:Npn \_text_expand_math_search:NNN #1#2#3
33051 {
33052   \_text_if_q_recursion_tail_stop_do:Nn #2
33053   { \_text_expand_explicit:N #1 }
33054   \token_if_eq_meaning:NNTF #1 #2
33055   {
33056     \_text_use_i_delimit_by_q_recursion_stop:nw
33057     {
33058       \_text_expand_store:n {#1}
33059       \_text_expand_math_loop:Nw #3
33060     }
33061   }
33062   { \_text_expand_math_search:NNN #1 }
33063 }
33064 \cs_new:Npn \_text_expand_math_loop:Nw #1#2 \s_text_recursion_stop
33065 {
33066   \tl_if_head_is_N_type:nTF {#2}
33067   { \_text_expand_math_N_type:NN }
33068   {
33069     \tl_if_head_is_group:nTF {#2}
33070     { \_text_expand_math_group:Nn }
33071     { \_text_expand_math_space:Nw }
33072   }
33073   #1#2 \s_text_recursion_stop
33074 }
33075 \cs_new:Npn \_text_expand_math_N_type:NN #1#2
33076 {
33077   \_text_if_s_recursion_tail_stop_do:Nn #2
33078   { \_text_expand_end:w }
33079   \token_if_eq_meaning:NNTF #2 \exp_not:N
33080   { \_text_expand_store:n {#2} }
33081   \token_if_eq_meaning:NNTF #2 #1
33082   { \_text_expand_loop:w }
33083   { \_text_expand_math_loop:Nw #1 }
33084 }
33085 \cs_new:Npn \_text_expand_math_group:Nn #1#2
33086 {
33087   \_text_expand_store:n { {#2} }
33088   \_text_expand_math_loop:Nw #1
33089 }
33090 \exp_after:wN \cs_new:Npn \exp_after:wN \_text_expand_math_space:Nw
33091 \exp_after:wN # \exp_after:wN 1 \c_space_tl
33092 {
33093   \_text_expand_store:n { ~ }
33094   \_text_expand_math_loop:Nw #1
33095 }

```

At this stage, either we have a control sequence or a simple character: split and handle. The need to check for non-protected actives arises from handling of legacy input encod-

ings: they need to end up in a representation we can deal with in further processing. The tests for explicit parts of the L^AT_EX 2_ε UTF-8 mechanism cover the case of bookmarks, where definitions change and are no longer protected. The same is true for **babel** shorthands. Ideally, all implicit tokens would be converted to their explicit equivalents here. However, that runs into issues with constructs that use `\chardef` or similar tokens. The one case we do convert is `\exp_stop_f:`, as that is predictable and should output a space.

```

33096 \cs_new:Npn \__text_expand_explicit:N #1
33097 {
33098   \token_if_cs:NTF #1
33099   { \__text_expand_exclude:N #1 }
33100   {
33101     \bool_lazy_and:nnTF
33102     { \token_if_active_p:N #1 }
33103     {
33104       ! \bool_lazy_any_p:n
33105       {
33106         { \token_if_protected_macro_p:N #1 }
33107         { \token_if_protected_long_macro_p:N #1 }
33108         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@two@octets }
33109         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@three@octets }
33110         { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@four@octets }
33111         { \tl_if_head_eq_meaning_p:oN {#1} \active@prefix }
33112       }
33113     }
33114     { \exp_after:wN \__text_expand_loop:w #1 }
33115     {
33116       \str_if_eq:nnTF {#1} { \exp_stop_f: }
33117       { \__text_expand_store:n { ~ } }
33118       { \__text_expand_store:n {#1} }
33119       \__text_expand_loop:w
33120     }
33121   }
33122 }

```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`. The switching command for case needs special handling as it has to work by meaning.

```

33123 \cs_new:Npn \__text_expand_exclude:N #1
33124 {
33125   \cs_if_eq:NNTF #1 \text_case_switch:nnnn
33126   { \__text_expand_exclude_switch:Nnnnn #1 }
33127   {
33128     \exp_args:Ne \__text_expand_exclude:nN
33129     {
33130       \exp_not:V \l_text_math_arg_tl
33131       \exp_not:V \l_text_expand_exclude_tl
33132       \exp_not:V \l_text_case_exclude_arg_tl
33133     }
33134     #1
33135   }
33136 }
33137 \cs_new:Npn \__text_expand_exclude_switch:Nnnnn #1#2#3#4#5
33138 {
33139   \__text_expand_store:n { #1 {#2} {#3} {#4} {#5} }

```

```

33140     \_text_expand_loop:w
33141   }
33142 \cs_new:Npn \_text_expand_exclude:nN #1#2
33143 {
33144     \_text_expand_exclude:NN #2 #1
33145     \q__text_recursion_tail \q__text_recursion_stop
33146 }
33147 \cs_new:Npn \_text_expand_exclude:NN #1#2
33148 {
33149     \_text_if_q_recursion_tail_stop_do:Nn #2
33150     { \_text_expand_accent:N #1 }
33151     \str_if_eq:nnTF {#1} {#2}
33152     {
33153         \_text_use_i_delimit_by_q_recursion_stop:nw
33154         { \_text_expand_exclude:Nw #1 }
33155     }
33156     { \_text_expand_exclude:NN #1 }
33157 }
33158 \cs_new:Npn \_text_expand_exclude:Nw #1#2#
33159 { \_text_expand_exclude:Nnn #1 {#2} }
33160 \cs_new:Npn \_text_expand_exclude:Nnn #1#2#3
33161 {
33162     \_text_expand_store:n { #1#2 {#3} }
33163     \_text_expand_loop:w
33164 }

```

Accents.

```

33165 \cs_new:Npn \_text_expand_accent:N #1
33166 {
33167     \exp_after:wN \_text_expand_accent:NN \exp_after:wN
33168     #1 \l_text_accents_tl
33169     \q__text_recursion_tail \q__text_recursion_stop
33170 }
33171 \cs_new:Npn \_text_expand_accent:NN #1#2
33172 {
33173     \_text_if_q_recursion_tail_stop_do:Nn #2
33174     { \_text_expand_letterlike:N #1 }
33175     \cs_if_eq:NNTF #2 #1
33176     {
33177         \_text_use_i_delimit_by_q_recursion_stop:nw
33178         {
33179             \_text_expand_store:n {#1}
33180             \_text_expand_loop:w
33181         }
33182     }
33183     { \_text_expand_accent:NN #1 }
33184 }

```

Another list of exceptions: these ones take no arguments so are easier to handle.

```

33185 \cs_new:Npn \_text_expand_letterlike:N #1
33186 {
33187     \exp_after:wN \_text_expand_letterlike:NN \exp_after:wN
33188     #1 \l_text_letterlike_tl
33189     \q__text_recursion_tail \q__text_recursion_stop
33190 }

```

```

33191 \cs_new:Npn \__text_expand_letterlike:NN #1#2
33192 {
33193   \__text_if_q_recursion_tail_stop_do:Nn #2
33194   { \__text_expand_cs:N #1 }
33195   \cs_if_eq:NNTF #2 #1
33196   {
33197     \__text_use_i_delimit_by_q_recursion_stop:nw
33198     {
33199       \__text_expand_store:n {#1}
33200       \__text_expand_loop:w
33201     }
33202   }
33203   { \__text_expand_letterlike:NN #1 }
33204 }

```

LaTeX 2_ε's `\protect` makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the `\protect` might be explicit, in which case we need to leave it alone. That includes the case where it's not even followed by an N-type token. There is also the case of a straight `\@protected@testopt` to cover.

```

33205 \cs_new:Npe \__text_expand_cs:N #1
33206 {
33207   \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
33208   { \exp_not:N \__text_expand_protect:w }
33209   {
33210     \bool_lazy_and:nnTF
33211     { \cs_if_exist_p:N \fmtname }
33212     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
33213     { \exp_not:N \__text_expand_testopt:N #1 }
33214     { \exp_not:N \__text_expand_replace:N #1 }
33215   }
33216 }
33217 \cs_new:Npn \__text_expand_protect:w #1 \s__text_recursion_stop
33218 {
33219   \tl_if_head_is_N_type:nTF {#1}
33220   { \__text_expand_protect:N }
33221   {
33222     \__text_expand_store:n { \protect }
33223     \__text_expand_loop:w
33224   }
33225   #1 \s__text_recursion_stop
33226 }
33227 \cs_new:Npn \__text_expand_protect:N #1
33228 {
33229   \__text_if_s_recursion_tail_stop_do:Nn #1
33230   {
33231     \__text_expand_store:n { \protect }
33232     \__text_expand_end:w
33233   }
33234   \exp_args:Ne \__text_expand_protect:nN
33235   { \cs_to_str:N #1 } #1
33236 }
33237 \cs_new:Npn \__text_expand_protect:nN #1#2
33238 { \__text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_stop
33239 \cs_new:Npn \__text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop

```

```

33240 {
33241   \__text_quark_if_nil:nTF {#4}
33242   {
33243     \cs_if_exist:cTF {#2}
33244     { \exp_args:Ne \__text_expand_store:n { \exp_not:c {#2} } }
33245     { \__text_expand_store:n { \protect #1 } }
33246   }
33247   { \__text_expand_store:n { \protect #1 } }
33248   \__text_expand_loop:w
33249 }
33250 \cs_new:Npn \__text_expand_testopt:N #1
33251 {
33252   \token_if_eq_meaning:NNTF #1 \@protected@testopt
33253   { \__text_expand_testopt:NNn }
33254   { \__text_expand_encoding:N #1 }
33255 }
33256 \cs_new:Npn \__text_expand_testopt:NNn #1#2#3
33257 {
33258   \__text_expand_store:n {#1}
33259   \__text_expand_loop:w
33260 }

```

Deal with encoding-specific commands

```

33261 \cs_new:Npn \__text_expand_encoding:N #1
33262 {
33263   \bool_lazy_or:nnTF
33264   { \cs_if_eq_p:NN #1 \@current@cmd }
33265   { \cs_if_eq_p:NN #1 \@changed@cmd }
33266   { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
33267   { \__text_expand_replace:N #1 }
33268 }
33269 \cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }

```

See if there is a dedicated replacement, and if there is, insert it.

```

33270 \cs_new:Npn \__text_expand_replace:N #1
33271 {
33272   \bool_lazy_and:nnTF
33273   { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _t1 } }
33274   {
33275     \bool_lazy_or_p:nn
33276     { \token_if_cs_p:N #1 }
33277     { \token_if_active_p:N #1 }
33278   }
33279   {
33280     \exp_args:Nv \__text_expand_replace:n
33281     { l__text_expand_ \token_to_str:N #1 _t1 }
33282   }
33283   { \__text_expand_cs_expand:N #1 }
33284 }
33285 \cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }

```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text.

```

33286 \cs_new:Npn \__text_expand_cs_expand:N #1

```



```

33287 {
33288   \__text_if_expandable:NTF #1
33289   {
33290     \token_if_eq_meaning:NNTF #1 \exp_not:n
33291     { \__text_expand_unexpanded:w }
33292     { \exp_after:wN \__text_expand_loop:w #1 }
33293   }
33294   {
33295     \__text_expand_store:n {#1}
33296     \__text_expand_loop:w
33297   }
33298 }

```

Since `\exp_not:n` is actually a primitive, it allows a strange syntax and it particular the primitive expands what follows and discards spaces and `\scan_stop:` until finding a braced argument (the opening brace can be implicit but we will not support this here). Here, we repeatedly `f`-expand after such an `\exp_not:n`, and test what follows. If it is a brace group, then we found the intended argument of `\exp_not:n`. If it is a space, then the next `f`-expansion will eliminate it. If it is an N-type token then `\__text_expand_unexpanded:N` leaves the token to be expanded if it is expandable, and otherwise removes it, assuming that it is `\scan_stop:`. This silently hides errors when `\exp_not:n` is incorrectly followed by some non-expandable token other than `\scan_stop:`, but this should be pretty rare, and there is no good error recovery anyways.

```

33299 \cs_new:Npn \__text_expand_unexpanded:w
33300 {
33301   \exp_after:wN \__text_expand_unexpanded_test:w
33302   \exp:w \exp_end_continue_f:w
33303 }
33304 \cs_new:Npn \__text_expand_unexpanded_test:w #1 \s__text_recursion_stop
33305 {
33306   \tl_if_head_is_group:nTF {#1}
33307   { \__text_expand_unexpanded:n }
33308   {
33309     \__text_expand_unexpanded:w
33310     \tl_if_head_is_N_type:nT {#1} { \__text_expand_unexpanded:N }
33311   }
33312   #1 \s__text_recursion_stop
33313 }
33314 \cs_new:Npn \__text_expand_unexpanded:N #1
33315 {
33316   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
33317   \else:
33318     \exp_after:wN #1
33319   \fi:
33320 }
33321 \cs_new:Npn \__text_expand_unexpanded:n #1
33322 {
33323   \__text_expand_store:n {#1}
33324   \__text_expand_loop:w
33325 }

```

(End of definition for `\text_expand:n` and others. This function is documented on page 303.)

`\text_declare_expand_equivalent:Nn` Create equivalents to allow replacement.
`\text_declare_expand_equivalent:cn`

```

33326 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
33327 {
33328   \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
33329   \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
33330 }
33331 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }

```

(End of definition for `\text_declare_expand_equivalent:Nn`. This function is documented on page [303](#).)

Prevent expansion of various standard values.

```

33332 \tl_map_inline:nn
33333 { \‘ \’ \^ \~ \= \u \. \. \r \H \v \d \c \k \b \t }
33334 { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
33335 \tl_map_inline:nn
33336 {
33337   \AA \aa
33338   \AE \ae
33339   \DH \dh
33340   \DJ \dj
33341   \IJ \ij
33342   \L \l
33343   \NG \ng
33344   \O \o
33345   \OE \oe
33346   \SS \ss
33347   \TH \th
33348 }
33349 { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
33350 </code>

```

Chapter 93

l3text-case implementation

```
33351 <*code>
33352 <@@=text>
```

93.1 Case changing

`\l_text_titlecase_check_letter_bool` Needed to determine the route used in titlecasing.

```
33353 \bool_new:N \l_text_titlecase_check_letter_bool
33354 \bool_set_true:N \l_text_titlecase_check_letter_bool
```

(End of definition for \l_text_titlecase_check_letter_bool. This variable is documented on page 306.)

`\text_lowercase:n` `\text_uppercase:n` The user level functions here are all wrappers around the internal functions for case changing.

```
\text_titlecase_all:n
\text_titlecase_first:n
\text_lowercase:nn
\text_lowercase:Vn
\text_uppercase:nn
\text_uppercase:Vn
\text_titlecase_all:nn
\text_titlecase_all:Vn
\text_titlecase_first:nn
\text_titlecase_first:Vn
__text_change_case:nnn
33355 \cs_new:Npn \text_lowercase:n #1
33356 { __text_change_case:nnn { lower } { } {#1} }
33357 \cs_new:Npn \text_uppercase:n #1
33358 { __text_change_case:nnn { upper } { } {#1} }
33359 \cs_new:Npn \text_titlecase_all:n #1
33360 { __text_change_case:nnn { title } { } {#1} }
33361 \cs_new:Npn \text_titlecase_first:n #1
33362 { __text_change_case:nnnn { title } { break } { } {#1} }
33363 \cs_new:Npn \text_lowercase:nn #1#2
33364 { __text_change_case:nnn { lower } {#1} {#2} }
33365 \cs_generate_variant:Nn \text_lowercase:nn { V }
33366 \cs_new:Npn \text_uppercase:nn #1#2
33367 { __text_change_case:nnn { upper } {#1} {#2} }
33368 \cs_generate_variant:Nn \text_uppercase:nn { V }
33369 \cs_new:Npn \text_titlecase_all:nn #1#2
33370 { __text_change_case:nnn { title } {#1} {#2} }
33371 \cs_generate_variant:Nn \text_titlecase_all:nn { V }
33372 \cs_new:Npn \text_titlecase_first:nn #1#2
33373 { __text_change_case:nnnn { title } { break } {#1} {#2} }
33374 \cs_generate_variant:Nn \text_titlecase_first:nn { V }
33375 \cs_new:Npn __text_change_case:nnn #1#2#3
33376 { __text_change_case:nnnn {#1} {#1} {#2} {#3} }
```

(End of definition for \text_lowercase:n and others. These functions are documented on page 304.)

```

\__text_change_case:nnnn
  \__text_change_case_auxi:nnnn
  \__text_change_case_auxii:nnnn
\__text_change_case_exclude_words:nn
  \__text_change_case_exclude_word:n
  \__text_change_case_inner:nnnn
\__text_change_case_bcp:nnn
  \__text_change_case_bcp:nnnnnnnn
  \__text_change_case_bcp:nnnnn
  \__text_change_case_loop:nnnw
\__text_change_case_break:
  \__text_change_case_group_lower:nnnn
  \__text_change_case_group_upper:nnnn
  \__text_change_case_group_title:nnnn
  \__text_change_case_space:nnnw
  \__text_change_case_space_break:nnn
  \__text_change_case_space_break:w
\__text_change_case_space_break_aux:w
  \__text_change_case_N_type:nnnN
  \__text_change_case_N_type_aux:nnnN
  \__text_change_case_N_type:nnnnN
\__text_change_case_math_search:nnnnNN
  \__text_change_case_math_loop:nnnNw
  \__text_change_case_math_N_type:nnnNN
  \__text_change_case_math_group:nnnNn
  \__text_change_case_math_space:nnnNw
  \__text_change_case_cs_check:nnnN
  \__text_change_case_exclude:nnnN
\__text_change_case_exclude_aux:nnnN
  \__text_change_case_exclude:nnnn
  \__text_change_case_exclude:nnnnN
  \__text_change_case_exclude:nnnNN
  \__text_change_case_exclude:nnnNw
  \__text_change_case_exclude:nnnNnn
  \__text_change_case_replace:nnnN
  \__text_change_case_replace:nnnn
  \__text_change_case_replace:vnnn
  \__text_change_case_switch:nnnN
\__text_change_case_switch_lower:nnnNnnnn
\__text_change_case_switch_upper:nnnNnnnn
\__text_change_case_switch_title:nnnNnnnn
\__text_change_case_skip:nnw
  \__text_change_case_skip_N_type:nnN
  \__text_change_case_skip_group:nnn
  \__text_change_case_skip_space:nnw
\__text_change_case_letterlike_lower:nnnN
\__text_change_case_letterlike_upper:nnnN
\__text_change_case_letterlike_title:nnnN
  \__text_change_case_letterlike:nnnnnN
  \__text_change_case_custom_lower:nnnn
  \__text_change_case_custom_title:nnnn
  \__text_change_case_custom_upper:nnnn
  \__text_change_case_custom:nnnnn
\__text_change_case_codepoint_lower:nnnn
\__text_change_case_codepoint_upper:nnnn
\__text_change_case_codepoint_title:nnnn
  \__text_change_case_lower_sigma:nnnnn
  \__text_change_case_lower_sigma:nnnnw
  \__text_change_case_lower_sigma:nnnnN
text change case codepoint title auxi:nnnn

```

As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to expand in a predictable fashion to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```

\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}

```

If we use an e-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, with a slightly strange mix of expansion control.

```

33377 \cs_new:Npn \__text_change_case:nnnn #1#2#3#4
33378 {
33379   \exp_not:e
33380   {
33381     \exp_args:Ne \__text_change_case_auxi:nnnn
33382     { \text_expand:n {#4} }
33383     {#1} {#2} {#3}
33384   }
33385 }
33386 \cs_new:Npn \__text_change_case_auxi:nnnn #1#2#3#4
33387 {
33388   \tl_if_blank:nTF {#4}
33389   { \__text_change_case_auxii:nnnn {#1} {#2} {#3} { } }
33390   {
33391     \exp_args:Ne \__text_change_case_bcp:nn
33392     { \text_bcp_parse:n {#4} } { {#1} {#2} {#3} }
33393   }
33394 }
33395 \cs_new:Npn \__text_change_case_bcp:nn #1#2
33396 { \__text_change_case_bcp:nnnnnnnn #1 {#2} }
33397 \cs_new:Npn \__text_change_case_bcp:nnnnnnnn #1#2#3#4#5#6#7#8
33398 {
33399   \tl_if_blank:nTF {#7}
33400   { \__text_change_case_auxii:nnnn #8 {#1} }
33401   {
33402     \__text_change_case_auxii:nnnn #8 {#1} #7
33403     \q_recursion_tail \q_recursion_stop
33404   }
33405 }
33406 \cs_new:Npn \__text_change_case_auxii:nnnn #1#2#3#4#5
33407 {
33408   \quark_if_recursion_tail_stop_do:nn {#5}
33409   { \__text_change_case_auxii:nnnn {#1} {#2} {#3} {#4} }
33410   \bool_lazy_or:nnT
33411   { \cs_if_exist_p:c { __text_change_case_ #2 _ #4 -x- #5 :nnnnn } }
33412   { \tl_if_exist_p:c { l__text_ #2 case_special_ #4 -x- #5 _tl } }
33413   {
33414     \use_i_delimit_by_q_recursion_stop:nw

```

```

33415         { \_text_change_case_auxii:nnnn {#1} {#2} {#3} { #4 -x- #5 } }
33416     }
33417     \_text_change_case_auxii:nnnn {#1} {#2} {#3} {#4}
33418 }

```

Dealing with excluded words is only really manageable if they are done in a separate loop. To avoid expanding twice, we use the internal. Other than that, this is basically a quick lookup.

```

33419 \cs_new:Npn \_text_change_case_auxii:nnnn #1#2#3#4
33420 {
33421     \group_align_safe_begin:
33422     \exp_args:Ne \_text_change_case_inner:nnnn
33423     {
33424         \_text_map_tokens:nnn {#1}
33425         { wordbreak } { \_text_change_case_exclude_words:nn {#2} }
33426     }
33427     {#2} {#3} {#4}
33428     \group_align_safe_end:
33429 }
33430 \cs_new:Npn \_text_change_case_exclude_words:nn #1#2
33431 {
33432     \tl_if_exist:cTF { l\_text_ #1 case_exclude_ \tl_to_str:n {#2} _tl }
33433     { \exp_not:n { \_text_change_case_exclude_word:n {#2} } }
33434     { \exp_not:n {#2} }
33435 }
33436 \cs_new:Npn \_text_change_case_exclude_word:n #1 {#1}
33437 \cs_new:Npn \_text_change_case_inner:nnnn #1#2#3#4
33438 {
33439     \cs_if_exist_use:c { \_text_change_case_boundary_ #2 _ #4 :Nnnnw }
33440     \_text_change_case_loop:nnnw {#2} {#3} {#4} #1
33441     \q__text_recursion_tail \q__text_recursion_stop
33442     \prg_break_point:Nn \_text_change_case_break: { }
33443 }

```

The main loop is the standard `tl` action type.

```

33444 \cs_new:Npn \_text_change_case_loop:nnnw #1#2#3#4 \q__text_recursion_stop
33445 {
33446     \tl_if_head_is_N_type:nTF {#4}
33447     { \_text_change_case_N_type:nnnN }
33448     {
33449         \tl_if_head_is_group:nTF {#4}
33450         { \use:c { \_text_change_case_group_ #1 :nnnn } }
33451         { \_text_change_case_space:nnnw }
33452     }
33453     {#1} {#2} {#3} #4 \q__text_recursion_stop
33454 }
33455 \cs_new:Npn \_text_change_case_break:
33456 { \prg_map_break:Nn \_text_change_case_break: { } }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

33457 \cs_new:Npn \__text_change_case_group_lower:nnnn #1#2#3#4
33458 {
33459   \exp_not:e
33460   { { \__text_change_case_inner:nnnn {#4} {#1} {#2} {#3} } }
33461   \__text_change_case_loop:nnnw {#1} {#2} {#3}
33462 }
33463 \cs_new_eq:NN \__text_change_case_group_upper:nnnn
33464   \__text_change_case_group_lower:nnnn
33465 \cs_new:Npn \__text_change_case_group_title:nnnn #1#2#3#4
33466 {
33467   \exp_not:e
33468   { { \__text_change_case_inner:nnnn {#4} {#1} {#2} {#3} } }
33469   \__text_change_case_skip:nnw {#2} {#3}
33470 }
33471 \use:e
33472 {
33473   \cs_new:Npn \exp_not:N \__text_change_case_space:nnnw #1#2#3 \c_space_tl
33474 }
33475 {
33476   \c_space_tl
33477   \cs_if_exist_use:cF { \__text_change_case_space_ #2 :nnn }
33478   {
33479     \cs_if_exist_use:c { \__text_change_case_boundary_ #1 _ #3 :Nnnnw }
33480     \__text_change_case_loop:nnnw
33481   }
33482   {#2} {#2} {#3}
33483 }
33484 \cs_new:Npn \__text_change_case_space_break:nnn #1#2#3
33485 { \__text_change_case_space_break:w \prg_do_nothing: }

```

Tidy up any protected words.

```

33486 \cs_new:Npn \__text_change_case_space_break:w
33487   #1 \q__text_recursion_tail \q__text_recursion_stop
33488 {
33489   \__text_change_case_space_break_aux:w #1
33490   \__text_change_case_exclude_word:n \q__text_recursion_tail \q__text_recursion_stop
33491 }
33492 \cs_new:Npn \__text_change_case_space_break_aux:w #1 \__text_change_case_exclude_word:n #2
33493 {
33494   \exp_not:o {#1}
33495   \__text_if_q_recursion_tail_stop_do:nn {#2} { \__text_change_case_break: }
33496   \__text_change_case_space_break_aux:w \prg_do_nothing: #2
33497 }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e., there is no assumption of “well-behaved” input in terms of math mode).

```

33498 \cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
33499 {
33500   \__text_if_q_recursion_tail_stop_do:Nn #4
33501   { \__text_change_case_break: }
33502   \__text_change_case_N_type_aux:nnnN {#1} {#2} {#3} #4
33503 }

```

```

33504 \cs_new:Npn \__text_change_case_N_type_aux:nnnN #1#2#3#4
33505 {
33506   \exp_args:NV \__text_change_case_N_type:nnnnN
33507   \l_text_math_delims_tl {#1} {#2} {#3} #4
33508 }
33509 \cs_new:Npn \__text_change_case_N_type:nnnnN #1#2#3#4#5
33510 {
33511   \__text_change_case_math_search:nnnNNN {#2} {#3} {#4} #5 #1
33512   \q__text_recursion_tail \q__text_recursion_tail
33513   \q__text_recursion_stop
33514 }
33515 \cs_new:Npn \__text_change_case_math_search:nnnNNN #1#2#3#4#5#6
33516 {
33517   \__text_if_q_recursion_tail_stop_do:Nn #5
33518   { \__text_change_case_cs_check:nnnN {#1} {#2} {#3} #4 }
33519   \token_if_eq_meaning:NNTF #4 #5
33520   {
33521     \__text_use_i_delimit_by_q_recursion_stop:nw
33522     {
33523       \exp_not:n {#4}
33524       \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #6
33525     }
33526   }
33527   { \__text_change_case_math_search:nnnNNN {#1} {#2} {#3} #4 }
33528 }
33529 \cs_new:Npn \__text_change_case_math_loop:nnnNw #1#2#3#4#5 \q__text_recursion_stop
33530 {
33531   \tl_if_head_is_N_type:nTF {#5}
33532   { \__text_change_case_math_N_type:nnnNN }
33533   {
33534     \tl_if_head_is_group:nTF {#5}
33535     { \__text_change_case_math_group:nnnNn }
33536     { \__text_change_case_math_space:nnnNw }
33537   }
33538   {#1} {#2} {#3} #4 #5 \q__text_recursion_stop
33539 }
33540 \cs_new:Npn \__text_change_case_math_N_type:nnnNN #1#2#3#4#5
33541 {
33542   \__text_if_q_recursion_tail_stop_do:Nn #5
33543   { \__text_change_case_break: }
33544   \exp_not:n {#5}
33545   \token_if_eq_meaning:NNTF #5 #4
33546   { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
33547   { \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4 }
33548 }
33549 \cs_new:Npn \__text_change_case_math_group:nnnNn #1#2#3#4#5
33550 {
33551   \exp_not:n { {#5} }
33552   \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
33553 }
33554 \use:e
33555 {
33556   \cs_new:Npn \exp_not:N \__text_change_case_math_space:nnnNw #1#2#3#4
33557   \c_space_tl

```

```

33558 }
33559 {
33560   \c_space_tl
33561   \_text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
33562 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

33563 \cs_new:Npn \_text_change_case_cs_check:nnnN #1#2#3#4
33564 {
33565   \token_if_cs:NTF #4
33566   { \_text_change_case_exclude:nnnN {#1} {#2} {#3} }
33567   {
33568     \_text_codepoint_process:nN
33569     { \use:c { \_text_change_case_custom_ #1 :nnnn } {#1} {#2} {#3} }
33570   }
33571   #4
33572 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

33573 \cs_new:Npn \_text_change_case_exclude:nnnN #1#2#3#4
33574 {
33575   \str_if_eq:nnTF {#4} { \_text_change_case_exclude_word:n }
33576   { \_text_change_case_exclude:nnnn {#1} {#2} {#3} }
33577   { \_text_change_case_exclude_aux:nnnN {#1} {#2} {#3} #4 }
33578 }
33579 \cs_new:Npn \_text_change_case_exclude_aux:nnnN #1#2#3#4
33580 {
33581   \exp_args:Ne \_text_change_case_exclude:nnnnN
33582   {
33583     \exp_not:V \l_text_math_arg_tl
33584     \exp_not:V \l_text_case_exclude_arg_tl
33585   }
33586   {#1} {#2} {#3} #4
33587 }

```

The internal special case function: simply grab the argument, protect and loop.

```

33588 \cs_new:Npn \_text_change_case_exclude:nnnn #1#2#3#4
33589 {
33590   \exp_not:n {#4}
33591   \_text_change_case_loop:nnnw {#1} {#2} {#3}
33592 }
33593 \cs_new:Npn \_text_change_case_exclude:nnnnN #1#2#3#4#5
33594 {
33595   \_text_change_case_exclude:nnnnN {#2} {#3} {#4} #5 #1
33596   \q__text_recursion_tail \q__text_recursion_stop
33597 }
33598 \cs_new:Npn \_text_change_case_exclude:nnnnN #1#2#3#4#5
33599 {
33600   \_text_if_q_recursion_tail_stop_do:Nn #5
33601   { \_text_change_case_replace:nnnN {#1} {#2} {#3} #4 }
33602   \str_if_eq:nnTF {#4} {#5}
33603   {

```



```

33604         \_text_use_i_delimit_by_q_recursion_stop:nw
33605         { \_text_change_case_exclude:nnnNw {#1} {#2} {#3} #4 }
33606     }
33607     { \_text_change_case_exclude:nnnNN {#1} {#2} {#3} #4 }
33608 }
33609 \cs_new:Npn \_text_change_case_exclude:nnnNw #1#2#3#4#5#
33610 { \_text_change_case_exclude:nnnNnn {#1} {#2} {#3} {#4} {#5} }
33611 \cs_new:Npn \_text_change_case_exclude:nnnNnn #1#2#3#4#5#6
33612 {
33613     \tl_if_blank:nTF {#5}
33614     { \exp_not:n { #4 {#6} } }
33615     {
33616         \exp_not:e
33617         {
33618             \exp_not:N #4
33619             \_text_change_case_inner:nnnn {#5} {#1} {#2} {#3}
33620             {#6}
33621         }
33622     }
33623     \_text_change_case_loop:nnnw {#1} {#2} {#3}
33624 }

```

Deal with any specialist replacement for case changing.

```

33625 \cs_new:Npn \_text_change_case_replace:nnnN #1#2#3#4
33626 {
33627     \cs_if_exist:cTF { l__text_case_ \token_to_str:N #4 _tl }
33628     {
33629         \_text_change_case_replace:vnnn
33630         { l__text_case_ \token_to_str:N #4 _tl } {#1} {#2} {#3}
33631     }
33632     { \_text_change_case_switch:nnnN {#1} {#2} {#3} #4 }
33633 }
33634 \cs_new:Npn \_text_change_case_replace:nnnn #1#2#3#4
33635 { \_text_change_case_loop:nnnw {#2} {#3} {#4} #1 }
33636 \cs_generate_variant:Nn \_text_change_case_replace:nnnn { v }

```

Allow for manually-controlled case switching.

```

33637 \cs_new:Npn \_text_change_case_switch:nnnN #1#2#3#4
33638 {
33639     \cs_if_eq:NNTF #4 \text_case_switch:nnnn
33640     { \use:c { __text_change_case_switch_ #1 :nnnNnnnn } }
33641     { \use:c { __text_change_case_letterlike_ #1 :nnnN } }
33642     {#1} {#2} {#3} #4
33643 }
33644 \cs_new:Npn \_text_change_case_switch_lower:nnnNnnnn #1#2#3#4#5#6#7#8
33645 {
33646     \exp_not:n {#7}
33647     \_text_change_case_loop:nnnw {#1} {#2} {#3}
33648 }
33649 \cs_new:Npn \_text_change_case_switch_upper:nnnNnnnn #1#2#3#4#5#6#7#8
33650 {
33651     \exp_not:n {#6}
33652     \_text_change_case_loop:nnnw {#1} {#2} {#3}
33653 }
33654 \cs_new:Npn \_text_change_case_switch_title:nnnNnnnn #1#2#3#4#5#6#7#8

```

```

33655 {
33656   \exp_not:n {#8}
33657   \__text_change_case_skip:nnw {#2} {#3}
33658 }

```

Skip over material quickly after titlecase-first-only initials

```

33659 \cs_new:Npn \__text_change_case_skip:nnw #1#2#3 \q__text_recursion_stop
33660 {
33661   \tl_if_head_is_N_type:nTF {#3}
33662   { \__text_change_case_skip_N_type:nnN }
33663   {
33664     \tl_if_head_is_group:nTF {#3}
33665     { \__text_change_case_skip_group:nnn }
33666     { \__text_change_case_skip_space:nnw }
33667   }
33668   {#1} {#2} #3 \q__text_recursion_stop
33669 }
33670 \cs_new:Npn \__text_change_case_skip_N_type:nnN #1#2#3
33671 {
33672   \__text_if_q_recursion_tail_stop_do:Nn #3
33673   { \__text_change_case_break: }
33674   \exp_not:n {#3}
33675   \__text_change_case_skip:nnw {#1} {#2}
33676 }
33677 \cs_new:Npn \__text_change_case_skip_group:nnn #1#2#3
33678 {
33679   \exp_not:n { {#3} }
33680   \__text_change_case_skip:nnw {#1} {#2}
33681 }
33682 \cs_new:Npn \__text_change_case_skip_space:nnw #1#2
33683 { \__text_change_case_space:nnnw {#1} {#1} {#2} }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

33684 \cs_new:Npn \__text_change_case_letterlike_lower:nnnN #1#2#3#4
33685 { \__text_change_case_letterlike:nnnnN {#1} {#1} {#1} {#2} {#3} #4 }
33686 \cs_new_eq:NN \__text_change_case_letterlike_upper:nnnN
33687   \__text_change_case_letterlike_lower:nnnN
33688 \cs_new:Npn \__text_change_case_letterlike_title:nnnN #1#2#3#4
33689 { \__text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} {#3} #4 }
33690 \cs_new:Npn \__text_change_case_letterlike:nnnnN #1#2#3#4#5#6
33691 {
33692   \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #6 _tl }
33693   {
33694     \exp_not:v
33695     { c__text_ #1 case_ \token_to_str:N #6 _tl }
33696     \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5}
33697   }
33698   {
33699     \exp_not:n {#6}
33700     \cs_if_exist:cTF
33701     {
33702       c__text_

```

```

33703         \str_if_eq:nnTF {#1} { lower } { upper } { lower }
33704         case_ \token_to_str:N #6 _tl
33705     }
33706     { \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5} }
33707     { \__text_change_case_loop:nnnw {#3} {#4} {#5} }
33708 }
33709 }

```

Check for a customized codepoint result.

```

33710 \cs_new:Npn \__text_change_case_custom_lower:nnnn #1#2#3#4
33711 {
33712     \__text_change_case_custom:nnnnnn {#1} {#1} {#2} {#3} {#4}
33713     { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
33714 }
33715 \cs_new_eq:NN \__text_change_case_custom_upper:nnnn
33716     \__text_change_case_custom_lower:nnnn
33717 \cs_new:Npn \__text_change_case_custom_title:nnnn #1#2#3#4
33718 {
33719     \__text_change_case_custom:nnnnnn { title } {#1} {#2} {#3} {#4}
33720     {
33721         \__text_change_case_custom:nnnnnn { upper } {#1} {#2} {#3} {#4}
33722         { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
33723     }
33724 }
33725 \cs_new:Npn \__text_change_case_custom:nnnnnn #1#2#3#4#5#6
33726 {
33727     \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl }
33728     {
33729         \__text_change_case_replace:vnnn
33730         { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl } {#2} {#3} {#4}
33731     }
33732     {
33733         \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _tl }
33734         {
33735             \__text_change_case_replace:vnnn
33736             { l__text_ #1 case _ \tl_to_str:n {#5} _tl } {#2} {#3} {#4}
33737         }
33738         {#6}
33739     }
33740 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple codepoint mapping.

```

33741 \cs_new:Npn \__text_change_case_codepoint_lower:nnnn #1#2#3#4
33742 {
33743     \cs_if_exist_use:cF { __text_change_case_lower_ #3 :nnnnn }
33744     { \__text_change_case_lower_sigma:nnnnn }
33745     {#1} {#1} {#2} {#3} {#4}
33746 }
33747 \cs_new:Npn \__text_change_case_codepoint_upper:nnnn #1#2#3#4
33748 {
33749     \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnnn }
33750     { \__text_change_case_codepoint:nnnnn }
33751     {#1} {#1} {#2} {#3} {#4}

```

```
33752 }
```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters or actives (to cover 8-bit engines).

```
33753 \cs_new:Npn \__text_change_case_lower_sigma:nnnnn #1#2#3#4#5
33754 {
33755   \__text_codepoint_compare:nNnTF {#5} = { "03A3 }
33756   { \__text_change_case_lower_sigma:nnnnw {#2} }
33757   { \__text_change_case_codepoint:nnnnn {#1} {#2} }
33758   {#3} {#4} {#5}
33759 }
33760 \cs_new:Npn \__text_change_case_lower_sigma:nnnnw #1#2#3#4#5 \q_text_recursion_stop
33761 {
33762   \tl_if_head_is_N_type:nTF {#5}
33763   { \__text_change_case_lower_sigma:nnnnN {#4} }
33764   {
33765     \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #4 }
33766     \__text_change_case_loop:nnnw
33767   }
33768   {#1} {#2} {#3} #5 \q_text_recursion_stop
33769 }
33770 \cs_new:Npn \__text_change_case_lower_sigma:nnnnN #1#2#3#4#5
33771 {
33772   \bool_lazy_or:nnTF
33773   { \token_if_letter_p:N #5 }
33774   {
33775     \bool_lazy_and_p:nn
33776     { \token_if_active_p:N #5 }
33777     { \int_compare_p:nNn {'#5} > { "80 } }
33778   }
33779   { \codepoint_generate:nn { "03C3 } { \__text_char_catcode:N #1 } }
33780   { \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #1 } }
33781   \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
33782 }
```

For titlecasing, we need to obtain the general category of the current codepoint.

```
33783 \cs_new:Npn \__text_change_case_codepoint_title:nnnn #1#2#3#4
33784 {
33785   \bool_if:NTF \l_text_titlecase_check_letter_bool
33786   {
33787     \exp_args:Ne \__text_change_case_codepoint_title_auxi:nnnn
33788     {
33789       \codepoint_to_category:n
33790       { \__text_codepoint_from_chars:Nw #4 }
33791     }
33792   }
33793   { \__text_change_case_codepoint_title:nnn }
33794   {#2} {#3} {#4}
33795 }
33796 \cs_new:Npn \__text_change_case_codepoint_title_auxi:nnnn #1#2#3#4
33797 {
33798   \tl_if_head_eq_charcode:nNTF {#1} { L }
33799   { \__text_change_case_codepoint_title:nnn }
33800   { \__text_change_case_codepoint_title_auxii:nnnn { title } }
```

```

33801         {#2} {#3} {#4}
33802     }
33803 \cs_new:Npn \__text_change_case_codepoint_title:nnn #1#2#3
33804 { \__text_change_case_codepoint_title_auxii:nnnn { end } {#1} {#2} {#3} }
33805 \cs_new:Npn \__text_change_case_codepoint_title_auxii:nnnn #1#2#3#4
33806 {
33807     \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnnn }
33808     {
33809         \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnnn }
33810         { \__text_change_case_codepoint:nnnnn }
33811     }
33812     { title } {#1} {#2} {#3} {#4}
33813 }
33814 \cs_new:Npn \__text_change_case_codepoint:nnnnn #1#2#3#4#5
33815 {
33816     \bool_lazy_and:nnTF
33817     { \tl_if_single_p:n {#5} }
33818     { \token_if_active_p:N #5 }
33819     { \exp_not:n {#5} }
33820     { \__text_change_case_codepoint:nn {#1} {#5} }
33821     \use:c { __text_change_case_next_ #2 :nnn } {#2} {#3} {#4}
33822 }
33823 \cs_new:Npn \__text_change_case_codepoint:nn #1#2
33824 {
33825     \__text_change_case_codepoint:fnn
33826     { \int_eval:n { \__text_codepoint_from_chars:Nw #2 } } {#1} {#2}
33827 }
33828 \cs_new:Npn \__text_change_case_codepoint:nnn #1#2#3
33829 {
33830     \exp_args:Ne \__text_change_case_codepoint_aux:nn
33831     { \__kernel_codepoint_case:nn { #2 case } {#1} } {#3}
33832 }
33833 \cs_generate_variant:Nn \__text_change_case_codepoint:nnn { f }
Avoid high chars with pTeX.
33834 \sys_if_engine_ptex:T
33835 {
33836     \cs_new_eq:NN \__text_change_case_codepoint_aux:nnn
33837     \__text_change_case_codepoint:nnn
33838     \cs_gset:Npn \__text_change_case_codepoint:nnn #1#2#3
33839     {
33840         \int_compare:nNnTF {#1} = { -1 }
33841         { \exp_not:n {#3} }
33842         { \__text_change_case_codepoint_aux:nnn {#1} {#2} {#3} }
33843     }
33844 }
33845 \cs_new:Npn \__text_change_case_codepoint_aux:nn #1#2
33846 {
33847     \use:e { \__text_change_case_codepoint_aux:nnnn #1 {#2} }
33848 }
33849 \cs_new:Npn \__text_change_case_codepoint_aux:nnnn #1#2#3#4
33850 {
33851     \__text_codepoint_compare:nNnTF {#4} = {#1}
33852     { \exp_not:n {#4} }
33853     {

```

```

33854     \codepoint_generate:nn {#1}
33855     { \__text_change_case_catcode:nn {#4} {#1} }
33856   \tl_if_blank:nF {#2}
33857   {
33858     \codepoint_generate:nn {#2}
33859     { \char_value_catcode:n {#2} }
33860     \tl_if_blank:nF {#3}
33861     {
33862       \codepoint_generate:nn {#3}
33863       { \char_value_catcode:n {#3} }
33864     }
33865   }
33866 }
33867 }

```

We need to ensure that only valid catcode-extraction is attempted. That's fine with Unicode engines but needs a bit of work with 8-bit ones. The logic is that if the original codepoint was in the ASCII range, we keep the catcode. Otherwise, if the target is in the ASCII range, we use the standard catcode. If neither are true, we set as 13 on the grounds that this will be what is used anyway!

```

33868 \sys_if_engine_opentype:TF
33869 {
33870   \cs_new:Npn \__text_change_case_catcode:nn #1#2
33871   { \__text_char_catcode:N #1 }
33872 }
33873 {
33874   \cs_new:Npn \__text_change_case_catcode:nn #1#2
33875   {
33876     \__text_codepoint_compare:nNnTF {#1} < { "80 }
33877     { \__text_char_catcode:N #1 }
33878     {
33879       \int_compare:nNnTF {#2} < { "80 }
33880       { \char_value_catcode:n {#2} }
33881       { 13 }
33882     }
33883   }
33884 }
33885 \cs_new:Npn \__text_change_case_next_lower:nnn #1#2#3
33886 { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
33887 \cs_new_eq:NN \__text_change_case_next_upper:nnn
33888 \__text_change_case_next_lower:nnn
33889 \cs_new_eq:NN \__text_change_case_next_title:nnn
33890 \__text_change_case_next_lower:nnn
33891 \cs_new:Npn \__text_change_case_next_end:nnn #1#2#3
33892 { \__text_change_case_skip:nnw {#2} {#3} }

```

(End of definition for __text_change_case:nnnn and others.)

`\text_declare_case_equivalent:Nn` Create equivalents to allow replacement.

```

33893 \cs_new_protected:Npn \text_declare_case_equivalent:Nn #1#2
33894 {
33895   \tl_clear_new:c { l__text_case_ \token_to_str:N #1 _tl }
33896   \tl_set:cn { l__text_case_ \token_to_str:N #1 _tl } {#2}
33897 }

```

(End of definition for `\text_declare_case_equivalent:Nn`. This function is documented on page 305.)

Codepoint customization.

```

\text_declare_lowercase_mapping:nn
\text_declare_titlecase_mapping:nn
\text_declare_uppercase_mapping:nn
  \_text_declare_case_mapping:nnn
\_text_declare_case_mapping_aux:nnn
\text_declare_lowercase_mapping:nnn
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nnn
  \_text_declare_case_mapping:nnnn
\_text_declare_case_mapping_aux:nnnn
33898 \cs_new_protected:Npn \text_declare_lowercase_mapping:nn #1#2
33899   { \_text_declare_case_mapping:nnn { lower } {#1} {#2} }
33900 \cs_new_protected:Npn \text_declare_titlecase_mapping:nn #1#2
33901   { \_text_declare_case_mapping:nnn { title } {#1} {#2} }
33902 \cs_new_protected:Npn \text_declare_uppercase_mapping:nn #1#2
33903   { \_text_declare_case_mapping:nnn { upper } {#1} {#2} }
33904 \cs_new_protected:Npn \_text_declare_case_mapping:nnn #1#2#3
33905   {
33906     \exp_args:Ne \_text_declare_case_mapping_aux:nnn
33907       { \codepoint_str_generate:n {#2} } {#1} {#3}
33908   }
33909 \cs_new_protected:Npn \_text_declare_case_mapping_aux:nnn #1#2#3
33910   {
33911     \tl_clear_new:c { l__text_ #2 case _ #1 _tl }
33912     \tl_set:cn { l__text_ #2 case _ #1 _tl } {#3}
33913   }
33914 \cs_new_protected:Npn \text_declare_lowercase_mapping:nnn #1#2#3
33915   { \_text_declare_case_mapping:nnnn { lower } {#1} {#2} {#3} }
33916 \cs_new_protected:Npn \text_declare_titlecase_mapping:nnn #1#2#3
33917   { \_text_declare_case_mapping:nnnn { title } {#1} {#2} {#3} }
33918 \cs_new_protected:Npn \text_declare_uppercase_mapping:nnn #1#2#3
33919   { \_text_declare_case_mapping:nnnn { upper } {#1} {#2} {#3} }
33920 \cs_new_protected:Npn \_text_declare_case_mapping:nnnn #1#2#3#4
33921   {
33922     \exp_args:Ne \_text_declare_case_mapping_aux:nnnn
33923       { \codepoint_str_generate:n {#3} } {#1} {#2} {#4}
33924   }
33925 \cs_new_protected:Npn \_text_declare_case_mapping_aux:nnnn #1#2#3#4
33926   {
33927     \tl_clear_new:c { l__text_ #2 case _ #1 _ #3 _tl }
33928     \tl_set:cn { l__text_ #2 case _ #1 _ #3 _tl } {#4}
33929     \tl_clear_new:c { l__text_ #2 case_special_ #3 _tl }
33930   }

```

(End of definition for `\text_declare_lowercase_mapping:nn` and others. These functions are documented on page 305.)

Prevent case change.

```

\text_declare_lowercase_exclusion:n
\text_declare_titlecase_exclusion:n
\text_declare_uppercase_exclusion:n
  \_text_declare_case_exclusion:nn
33931 \cs_new_protected:Npn \text_declare_lowercase_exclusion:n #1
33932   { \_text_declare_case_exclusion:nn { lower } {#1} }
33933 \cs_new_protected:Npn \text_declare_titlecase_exclusion:n #1
33934   { \_text_declare_case_exclusion:nn { title } {#1} }
33935 \cs_new_protected:Npn \text_declare_uppercase_exclusion:n #1
33936   { \_text_declare_case_exclusion:nn { upper } {#1} }
33937 \cs_new_protected:Npn \_text_declare_case_exclusion:nn #1#2
33938   { \tl_clear_new:c { l__text_ #1 case_exclude_ \tl_to_str:n {#2} _tl } }

```

(End of definition for `\text_declare_lowercase_exclusion:n` and others. These functions are documented on page 305.)

Set up the mechanism for manual case switching.

```

\text_case_switch:nnnn
\_text_case_switch_marker:
33939 \cs_new:Npn \text_case_switch:nnnn #1#2#3#4

```

```

33940 {
33941   \_text\_case\_switch\_marker:
33942   #1
33943 }
33944 \cs\_new:Npn \_text\_case\_switch\_marker: { }

```

(End of definition for \text_case_switch:nnnn and _text_case_switch_marker:. This function is documented on page 306.)

_text_change_case_generate:n

A utility.

```

33945 \cs\_new:Npn \_text\_change\_case\_generate:n #1
33946 { \codepoint\_generate:nn {#1} { \char\_value\_catcode:n {#1} } }

```

(End of definition for _text_change_case_generate:n.)

_text_change_case_upper_de-x-eszett:nnnnn

A simple alternative version for German.

_text_change_case_upper_de-alt:nnnnn

```

33947 \cs\_new:cpn { \_text\_change\_case\_upper\_de-x-eszett:nnnnn } #1#2#3#4#5
33948 {
33949   \_text\_codepoint\_compare:nNnTF {#5} = { "00DF }
33950   {
33951     \codepoint\_generate:nn { "1E9E }
33952     { \_text\_change\_case\_catcode:nn {#5} { "1E9E } }
33953     \use:c { \_text\_change\_case\_next\_ #2 :nnn }
33954     {#2} {#3} {#4}
33955   }
33956   { \_text\_change\_case\_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33957 }
33958 \cs\_new\_eq:cc { \_text\_change\_case\_upper\_de-alt:nnnnn }
33959 { \_text\_change\_case\_upper\_de-x-eszett:nnnnn }

```

(End of definition for _text_change_case_upper_de-x-eszett:nnnnn and _text_change_case_upper_de-alt:nnnnn.)

_text_change_case_upper_el:nnnnn

For Greek uppercasing, we need to know if characters *in the Greek range* have accents.

_text_change_case_upper_el-x-iota:nnnnn

That means doing a NFD conversion first, then starting a search. As described by the

_text_change_case_upper_el_aux:nnnnn

Unicode CLDR, Greek accents need to be found *after* any U+0308 (diaeresis) and are done

_text_change_case_upper_el:nnnn

in two groups to allow for the canonical ordering. The implementation here follows the

_text_change_case_upper_el:nnnnw

data and examples from ICU (<https://icu.unicode.org/design/case/greek-upper>),

_text_change_case_upper_el:nnnnN

although necessarily the implementation is somewhat different. The *ypogegrammeni* is

_text_change_case_upper_el_aux:nnnnN

filtered out here as it is not actually in the Greek range, so gets lost if we leave until later.

_text_change_case_upper_el_ypogegrammeni:nnnnnnw

The one Greek codepoint we skip is the numeral sign and question mark: the first has an

_text_change_case_upper_el_ypogegrammeni:nnnnnnN

awkward NFD for pdf_{TeX} so is best left unchanged, and the latter has issues concerning

_text_change_case_upper_el_ypogegrammeni:nnnnnnn

how LGR outputs the input and output (differently!).

_text_change_case_upper_el_dialytika:nnnn

```

33960 \cs\_new:Npn \_text\_change\_case\_upper\_el:nnnnn #1#2#3#4#5

```

_text_change_case_upper_el_dialytika:n

```

33961 {
33962   \bool\_lazy\_and:nnTF
33963   { \_text\_change\_case\_if\_greek\_p:n {#5} }
33964   {
33965     ! \bool\_lazy\_or\_p:nn
33966     { \_text\_codepoint\_compare\_p:nNn {#5} = { "0374 } }
33967     { \_text\_codepoint\_compare\_p:nNn {#5} = { "037E } }
33968   }
33969   {
33970     \_text\_change\_case\_if\_greek\_spacing\_diacritic:nTF {#5}

```



```

33971         {
33972             \exp_not:n {#5}
33973             \__text_change_case_loop:nnnw
33974         }
33975         {
33976             \exp_args:Ne \__text_change_case_upper_el:nnnn
33977             {
33978                 \codepoint_to_nfd:n
33979                 { \__text_codepoint_from_chars:Nw #5 }
33980             }
33981         }
33982         {#2} {#3} {#4}
33983     }
33984     {
33985         \__text_codepoint_compare:nNnTF {#5} = { "0345 }
33986         {
33987             \codepoint_generate:nn { "0399 }
33988             { \char_value_catcode:n { "0399 } }
33989             \__text_change_case_loop:nnnw {#2} {#3} {#4}
33990         }
33991         { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
33992     }
33993 }
33994 \cs_new_eq:cN { __text_change_case_upper_el-x-iota:nnnnn }
33995 \__text_change_case_upper_el:nnnnn
33996 \cs_new:Npn \__text_change_case_upper_el:nnnn #1#2#3#4
33997 {
33998     \__text_codepoint_process:nN
33999     { \__text_change_case_upper_el:nnnnw {#2} {#3} {#4} } #1
34000 }

```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```

34001 \cs_new:Npn \__text_change_case_upper_el:nnnnw #1#2#3#4#5 \q__text_recursion_stop
34002 {
34003     \tl_if_head_is_N_type:nTF {#5}
34004     { \__text_change_case_upper_el:nnnnN {#4} }
34005     {
34006         \__text_change_case_codepoint:nn { upper } {#4}
34007         \__text_change_case_loop:nnnw
34008     }
34009     {#1} {#2} {#3} #5 \q__text_recursion_stop
34010 }

```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.) There is additional work if the codepoint can take a ypogegrammeni: there, we need to move any ypogegrammeni to after accents (in case the input is not normalized). The ypogegrammeni itself is handled separately.

```

34011 \cs_new:Npn \__text_change_case_upper_el:nnnnN #1#2#3#4#5
34012 {
34013     \token_if_cs:NTF #5
34014     {

```

```

34015     \_text_change_case_codepoint:nn { upper } {#1}
34016     \_text_change_case_loop:nnnw {#2} {#3} {#4} #5
34017 }
34018 {
34019     \_text_change_case_if_takes_ypogegrammeni:nTF {#1}
34020     {
34021         \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
34022         {#1} {#2} {#3} {#4} { } { } #5
34023     }
34024     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5 }
34025 }
34026 }
34027 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
34028 #1#2#3#4#5#6#7 \q_text_recursion_stop
34029 {
34030     \tl_if_head_is_N_type:nTF {#7}
34031     {
34032         \_text_change_case_upper_el_ypogegrammeni:nnnnnnN
34033         {#1} {#2} {#3} {#4} {#5} {#6}
34034     }
34035     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
34036     #7 \q_text_recursion_stop
34037 }
34038 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnN #1#2#3#4#5#6#7
34039 {
34040     \token_if_cs:nTF #7
34041     { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
34042     {
34043         \_text_codepoint_process:nN
34044         {
34045             \_text_change_case_upper_el_ypogegrammeni:nnnnnnn
34046             {#1} {#2} {#3} {#4} {#5} {#6}
34047         }
34048     }
34049     #7
34050 }
34051 \cs_new:Npn \_text_change_case_upper_el_ypogegrammeni:nnnnnnn #1#2#3#4#5#6#7
34052 {
34053     \_text_codepoint_compare:nNnTF {#7} = { "0345 }
34054     {
34055         \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
34056         {#1} {#2} {#3} {#4} {#5} {#7}
34057     }
34058     {
34059         \bool_lazy_or:nnTF
34060         { \_text_change_case_if_greek_accent_p:n {#7} }
34061         { \_text_change_case_if_greek_breathing_p:n {#7} }
34062         {
34063             \_text_change_case_upper_el_ypogegrammeni:nnnnnnw
34064             {#1} {#2} {#3} {#4} {#5#7} {#6}
34065         }
34066         { \_text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 #7 }
34067     }
34068 }

```

```

34069 \cs_new:Npn \__text_change_case_upper_el_aux:nnnnN #1#2#3#4#5
34070 {
34071     \__text_codepoint_process:nN
34072     { \__text_change_case_upper_el_aux:nnnnn {#1} {#2} {#3} {#4} } #5
34073 }
34074 \cs_new:Npn \__text_change_case_upper_el_aux:nnnnn #1#2#3#4#5
34075 {
34076     \__text_codepoint_compare:nNnTF {#5} = { "0308 }
34077     { \__text_change_case_upper_el_dialytika:nnnn {#2} {#3} {#4} {#1} }
34078     {
34079         \__text_change_case_if_greek_accent:nTF {#5}
34080         { \__text_change_case_upper_el_hiatus:nnnnw {#2} {#3} {#4} {#1} }
34081         {
34082             \__text_change_case_if_greek_breathing:nTF {#5}
34083             { \__text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} }
34084             {
34085                 \__text_codepoint_compare:nNnTF {#5} = { "0345 }
34086                 {
34087                     \use:c { __text_change_case_upper_ #4 _ypogegrammeni:n } {#1}
34088                     \__text_change_case_loop:nnnw {#2} {#3} {#4}
34089                 }
34090                 {
34091                     \__text_change_case_if_greek_stress:nTF {#5}
34092                     {
34093                         \__text_change_case_upper_el_stress:nn {#1} {#5}
34094                         \__text_change_case_loop:nnnw {#2} {#3} {#4}
34095                     }
34096                     {
34097                         \__text_change_case_codepoint:nn { upper } {#1}
34098                         \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
34099                     }
34100                 }
34101             }
34102         }
34103     }
34104 }

```

We handle *dialytika* in parts as it's also needed for the hiatus. We know only two letters take it, so we can shortcut here on the second part of the tests.

```

34105 \cs_new:Npn \__text_change_case_upper_el_dialytika:nnnn #1#2#3#4
34106 {
34107     \__text_change_case_if_takes_dialytika:nTF {#4}
34108     { \__text_change_case_upper_el_dialytika:n {#4} }
34109     { \__text_change_case_codepoint:nn { upper } {#4} }
34110     \__text_change_case_upper_el_gobble:nnnw {#1} {#2} {#3}
34111 }
34112 \cs_new:Npn \__text_change_case_upper_el_dialytika:n #1
34113 {
34114     \bool_lazy_or:nnTF
34115     { \__text_codepoint_compare_p:nNn {#1} = { "0399 } }
34116     { \__text_codepoint_compare_p:nNn {#1} = { "03B9 } }
34117     {
34118         \codepoint_generate:nn { "03AA }
34119         { \__text_change_case_catcode:nn {#1} { "03AA } }

```

```

34120     }
34121     {
34122         \codepoint_generate:nn { "03AB }
34123         { \__text_change_case_catcode:nn {#1} { "03AB } }
34124     }
34125 }

```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```

34126 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnw
34127     #1#2#3#4#5 \q__text_recursion_stop
34128     {
34129         \tl_if_head_is_N_type:nTF {#5}
34130         { \__text_change_case_upper_el_hiatus:nnnnN {#4} }
34131         {
34132             \__text_change_case_codepoint:nn { upper } {#4}
34133             \__text_change_case_loop:nnnw
34134         }
34135         {#1} {#2} {#3} #5 \q__text_recursion_stop
34136     }
34137 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnN #1#2#3#4#5
34138     {
34139         \token_if_cs:NTF #5
34140         {
34141             \__text_change_case_codepoint:nn { upper } {#1}
34142             \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
34143         }
34144         {
34145             \__text_codepoint_process:nN
34146             { \__text_change_case_upper_el_hiatus:nnnnn {#1} {#2} {#3} {#4} } #5
34147         }
34148     }
34149 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnn #1#2#3#4#5
34150     {
34151         \__text_change_case_if_takes_dialytika:nTF {#5}
34152         {
34153             \__text_change_case_codepoint:nn { upper } {#1}
34154             \__text_change_case_upper_el_dialytika:n {#5}
34155             \__text_change_case_upper_el_gobble:nnnw {#2} {#3} {#4}
34156         }
34157         { \__text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} #5 }
34158     }

```

Handling the *ypogegrammeni* output depends on the selected approach

```

34159 \cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:n #1
34160     {
34161         \exp_args:Ne \__text_change_case_generate:n
34162         {
34163             \int_case:nn
34164             { \__text_codepoint_from_chars:Nw #1 }
34165             {
34166                 { "0391 } { "1FBC }
34167                 { "03B1 } { "1FBC }
34168                 { "0397 } { "1FCC }
34169                 { "03B7 } { "1FCC }

```

```

34170         { "03A9 } { "1FFC }
34171         { "03C9 } { "1FFC }
34172     }
34173 }
34174 }
34175 \cs_new:cpn { __text_change_case_upper_el-x-iota_ypogegrammeni:n } #1
34176 {
34177     \__text_change_case_codepoint:nn { upper } {#1}
34178     \codepoint_generate:nn { "0399 }
34179     { \char_value_catcode:n { "0399 } }
34180 }

```

We choose to retain stress diacritics, but we also need to recombine them for pdfTeX. That is handled here.

```

34181 \cs_new:Npn \__text_change_case_upper_el_stress:nn #1#2
34182 {
34183     \exp_args:Ne \__text_change_case_generate:n
34184     {
34185         \int_case:nn
34186         { \__text_codepoint_from_chars:Nw #2 }
34187         {
34188             { "0304 }
34189             {
34190                 \int_case:nn { \__text_codepoint_from_chars:Nw #1 }
34191                 {
34192                     { "0391 } { "1FB9 }
34193                     { "03B1 } { "1FB9 }
34194                     { "0399 } { "1FD9 }
34195                     { "03B9 } { "1FD9 }
34196                     { "03A5 } { "1FE9 }
34197                     { "03C5 } { "1FE9 }
34198                 }
34199             }
34200             { "0306 }
34201             {
34202                 \int_case:nn { \__text_codepoint_from_chars:Nw #1 }
34203                 {
34204                     { "0391 } { "1FB8 }
34205                     { "03B1 } { "1FB8 }
34206                     { "0399 } { "1FD8 }
34207                     { "03B9 } { "1FD8 }
34208                     { "03A5 } { "1FE8 }
34209                     { "03C5 } { "1FE8 }
34210                 }
34211             }
34212         }
34213     }
34214 }

```

For clearing out trailing combining marks after we have dealt with the first one.

```

34215 \cs_new:Npn \__text_change_case_upper_el_gobble:nnnw
34216 #1#2#3#4 \q__text_recursion_stop
34217 {
34218     \tl_if_head_is_N_type:nTF {#4}
34219     { \__text_change_case_upper_el_gobble:nnnN }

```

```

34220     { \_text_change_case_loop:nnnw }
34221     {#1} {#2} {#3} #4 \q\_text_recursion_stop
34222 }
34223 \cs_new:Npn \_text_change_case_upper_el_gobble:nnnN #1#2#3#4
34224 {
34225     \token_if_cs:NTF #4
34226     { \_text_change_case_loop:nnnw {#1} {#2} {#3} }
34227     {
34228         \_text_codepoint_process:nN
34229         { \_text_change_case_upper_el_gobble:nnnn {#1} {#2} {#3} }
34230     }
34231     #4
34232 }
34233 \cs_new:Npn \_text_change_case_upper_el_gobble:nnnn #1#2#3#4
34234 {
34235     \bool_lazy_or:nnTF
34236     { \_text_change_case_if_greek_accent_p:n {#4} }
34237     { \_text_change_case_if_greek_breathing_p:n {#4} }
34238     { \_text_change_case_upper_el_gobble:nnnw {#1} {#2} {#3} }
34239     { \_text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34240 }

```

Luckily the Greek range is limited and clear.

```

34241 \prg_new_conditional:Npnn \_text_change_case_if_greek:n #1 { p , TF }
34242 {
34243     \exp_args:Nf \_text_change_case_if_greek:n
34244     { \int_eval:n { \_text_codepoint_from_chars:Nw #1 } }
34245 }
34246 \cs_new:Npn \_text_change_case_if_greek:n #1
34247 {
34248     \if_int_compare:w #1 < "0370 \exp_stop_f:
34249     \prg_return_false:
34250     \else:
34251         \if_int_compare:w #1 > "03FF \exp_stop_f:
34252         \if_int_compare:w #1 < "1F00 \exp_stop_f:
34253         \prg_return_false:
34254         \else:
34255             \if_int_compare:w #1 > "1FFF \exp_stop_f:
34256             \if_int_compare:w #1 = "2126 \exp_stop_f:
34257             \prg_return_true:
34258             \else:
34259                 \prg_return_false:
34260                 \fi:
34261             \else:
34262                 \prg_return_true:
34263                 \fi:
34264             \fi:
34265         \else:
34266             \prg_return_true:
34267             \fi:
34268         \fi:
34269     }

```

We follow ICU in adding a few extras to the accent list here.

```

34270 \prg_new_conditional:Npnn \_text_change_case_if_greek_accent:n #1 { TF , p }

```

```

34271 {
34272   \exp_args:Nf \__text_change_case_if_greek_accent:n
34273   { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34274 }
34275 \cs_new:Npn \__text_change_case_if_greek_accent:n #1
34276 {
34277   \if_int_compare:w #1 = "0300 \exp_stop_f:
34278   \prg_return_true:
34279   \else:
34280     \if_int_compare:w #1 = "0301 \exp_stop_f:
34281     \prg_return_true:
34282     \else:
34283       \if_int_compare:w #1 = "0342 \exp_stop_f:
34284       \prg_return_true:
34285       \else:
34286         \if_int_compare:w #1 = "0302 \exp_stop_f:
34287         \prg_return_true:
34288         \else:
34289           \if_int_compare:w #1 = "0303 \exp_stop_f:
34290           \prg_return_true:
34291           \else:
34292             \if_int_compare:w #1 = "0311 \exp_stop_f:
34293             \prg_return_true:
34294             \else:
34295               \prg_return_false:
34296             \fi:
34297           \fi:
34298         \fi:
34299       \fi:
34300     \fi:
34301   \fi:
34302 }
34303 \prg_new_conditional:Npnn \__text_change_case_if_greek_spacing_diacritic:n
34304 #1 { TF }
34305 {
34306   \exp_args:Nf \__text_change_case_if_greek_spacing_diacritic:n
34307   { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34308 }
34309 \cs_new:Npn \__text_change_case_if_greek_spacing_diacritic:n #1
34310 {
34311   \if_int_compare:w #1 < "1FBD \exp_stop_f:
34312   \if_int_compare:w #1 = "037A \exp_stop_f:
34313   \prg_return_true:
34314   \else:
34315     \prg_return_false:
34316   \fi:
34317   \else:
34318     \if_int_compare:w #1 = "1FBD \exp_stop_f:
34319     \prg_return_true:
34320     \else:
34321       \if_int_compare:w #1 = "1FBF \exp_stop_f:
34322       \prg_return_true:
34323       \else:
34324         \if_int_compare:w #1 = "1FC0 \exp_stop_f:

```

```

34325         \prg_return_true:
34326     \else:
34327         \if_int_compare:w #1 = "1FC1 \exp_stop_f:
34328             \prg_return_true:
34329     \else:
34330         \if_int_compare:w #1 = "1FCD \exp_stop_f:
34331             \prg_return_true:
34332     \else:
34333         \if_int_compare:w #1 = "1FCE \exp_stop_f:
34334             \prg_return_true:
34335     \else:
34336         \if_int_compare:w #1 = "1FCF \exp_stop_f:
34337             \prg_return_true:
34338     \else:
34339         \if_int_compare:w #1 = "1FDD \exp_stop_f:
34340             \prg_return_true:
34341     \else:
34342         \if_int_compare:w #1 = "1FDE \exp_stop_f:
34343             \prg_return_true:
34344     \else:
34345         \if_int_compare:w #1 = "1FDF \exp_stop_f:
34346             \prg_return_true:
34347     \else:
34348         \if_int_compare:w #1 = "1FED \exp_stop_f:
34349             \prg_return_true:
34350     \else:
34351         \if_int_compare:w #1 = "1FEE \exp_stop_f:
34352             \prg_return_true:
34353     \else:
34354         \if_int_compare:w #1 = "1FEF \exp_stop_f:
34355             \prg_return_true:
34356     \else:
34357         \if_int_compare:w #1 = "1FFD \exp_stop_f:
34358             \prg_return_true:
34359     \else:
34360         \if_int_compare:w #1 = "1FFE \exp_stop_f:
34361             \prg_return_true:
34362         \else:
34363             \prg_return_false:
34364         \fi:
34365     \fi:
34366 \fi:
34367 \fi:
34368 \fi:
34369 \fi:
34370 \fi:
34371 \fi:
34372 \fi:
34373 \fi:
34374 \fi:
34375 \fi:
34376 \fi:
34377 \fi:
34378 \fi:

```



```

34379     \fi:
34380   }
34381 \prg_new_conditional:Npnn \__text_change_case_if_greek_breathing:n
34382   #1 { TF , p }
34383   {
34384     \exp_args:Nf \__text_change_case_if_greek_breathing:n
34385       { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34386   }
34387 \cs_new:Npn \__text_change_case_if_greek_breathing:n #1
34388   {
34389     \if_int_compare:w #1 = "0313 \exp_stop_f:
34390     \prg_return_true:
34391   \else:
34392     \if_int_compare:w #1 = "0314 \exp_stop_f:
34393     \prg_return_true:
34394   \else:
34395     \prg_return_false:
34396   \fi:
34397 \fi:
34398 }
34399 \prg_new_conditional:Npnn \__text_change_case_if_greek_stress:n
34400   #1 { TF , p }
34401   {
34402     \exp_args:Nf \__text_change_case_if_greek_stress:n
34403       { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34404   }
34405 \cs_new:Npn \__text_change_case_if_greek_stress:n #1
34406   {
34407     \if_int_compare:w #1 = "0304 \exp_stop_f:
34408     \prg_return_true:
34409   \else:
34410     \if_int_compare:w #1 = "0306 \exp_stop_f:
34411     \prg_return_true:
34412   \else:
34413     \prg_return_false:
34414   \fi:
34415 \fi:
34416 }
34417 \prg_new_conditional:Npnn \__text_change_case_if_takes_dialytika:n #1 { TF }
34418   {
34419     \exp_args:Nf \__text_change_case_if_takes_dialytika:n
34420       { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34421   }
34422 \cs_new:Npn \__text_change_case_if_takes_dialytika:n #1
34423   {
34424     \if_int_compare:w #1 = "0399 \exp_stop_f:
34425     \prg_return_true:
34426   \else:
34427     \if_int_compare:w #1 = "03B9 \exp_stop_f:
34428     \prg_return_true:
34429   \else:
34430     \if_int_compare:w #1 = "03A5 \exp_stop_f:
34431     \prg_return_true:
34432   \else:

```

```

34433         \if_int_compare:w #1 = "03C5 \exp_stop_f:
34434         \prg_return_true:
34435     \else:
34436         \prg_return_false:
34437     \fi:
34438 \fi:
34439 \fi:
34440 \fi:
34441 }
34442 \prg_new_conditional:Npnn \__text_change_case_if_takes_ypogegrammeni:n #1 { TF }
34443 {
34444     \exp_args:Nf \__text_change_case_if_takes_ypogegrammeni:n
34445     { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
34446 }
34447 \cs_new:Npn \__text_change_case_if_takes_ypogegrammeni:n #1
34448 {
34449     \if_int_compare:w #1 = "03B1 \exp_stop_f:
34450     \prg_return_true:
34451 \else:
34452     \if_int_compare:w #1 = "03B7 \exp_stop_f:
34453     \prg_return_true:
34454 \else:
34455     \if_int_compare:w #1 = "03C9 \exp_stop_f:
34456     \prg_return_true:
34457 \else:
34458     \prg_return_false:
34459 \fi:
34460 \fi:
34461 \fi:
34462 }

```

(End of definition for __text_change_case_upper_el:nnnnn and others.)

__text_change_case_boundary_upper_el:Nnnnw
 _change_case_boundary_upper_el-x-iota:Nnnnw
 __text_change_case_boundary_upper_el:nnnN
 __text_change_case_boundary_upper_el:nnnn
 __text_change_case_boundary_upper_el:nnnnw

There is one things that need special treatment at the start of words in Greek. For an isolated accent *eta*, which is handled by seeing if we have exactly one of the affected codepoints followed by a space or brace group.

```

34463 \cs_new:Npn \__text_change_case_boundary_upper_el:Nnnnw
34464     #1#2#3#4#5 \q__text_recursion_stop
34465 {
34466     \tl_if_head_is_N_type:nTF {#5}
34467     { \__text_change_case_boundary_upper_el:nnnN }
34468     { \__text_change_case_loop:nnnw }
34469     {#2} {#3} {#4} #5 \q__text_recursion_stop
34470 }
34471 \cs_new_eq:cN { __text_change_case_boundary_upper_el-x-iota:Nnnnw }
34472     \__text_change_case_boundary_upper_el:Nnnnw
34473 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnN #1#2#3#4
34474 {
34475     \token_if_cs:NTF #4
34476     { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
34477     {
34478         \__text_codepoint_process:nN
34479         { \__text_change_case_boundary_upper_el:nnnn {#1} {#2} {#3} }
34480     }

```

```

34481         #4
34482     }
34483 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnn #1#2#3#4
34484 {
34485     \bool_lazy_any:nTF
34486     {
34487         { \__text_codepoint_compare_p:nNn {#4} = { "0389 } }
34488         { \__text_codepoint_compare_p:nNn {#4} = { "03AE } }
34489         { \__text_codepoint_compare_p:nNn {#4} = { "1F22 } }
34490         { \__text_codepoint_compare_p:nNn {#4} = { "1F2A } }
34491     }
34492     { \__text_change_case_boundary_upper_el:nnnnw {#1} {#2} {#3} {#4} }
34493     { \__text_change_case_breathing:nnnn {#1} {#2} {#3} {#4} }
34494 }
34495 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnnw
34496 #1#2#3#4#5 \q_text_recursion_stop
34497 {
34498     \tl_if_head_is_N_type:nTF {#5}
34499     { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34500     {
34501         \codepoint_generate:nn { "0389 }
34502         { \__text_change_case_catcode:nn {#4} { "0389 } }
34503         \__text_change_case_loop:nnnw {#1} {#2} {#3}
34504     }
34505     #5 \q_text_recursion_stop
34506 }

```

(End of definition for __text_change_case_boundary_upper_el:Nnnnw and others.)

```

\__text_change_case_breathing:nnnn
\__text_change_case_breathing:nnnnn
\__text_change_case_breathing:nnnnnw
\__text_change_case_breathing:nnnnnnw
\__text_change_case_breathing_aux:nnnnnn
\__text_change_case_breathing_aux:nnnnnw
\__text_change_case_breathing_aux:nnnnN
\__text_change_case_breathing_dialytika:nnnn

```

In Greek, breathing diacritics are normally dropped when uppercasing: see the code for the general case. However, for the first character of a word, if there is a breather *and* the next character takes a *dialytika*, it needs to be added. We start by checking if the current codepoint is in the Greek range, then decomposing.

```

34507 \cs_new:Npn \__text_change_case_breathing:nnnn #1#2#3#4
34508 {
34509     \__text_change_case_if_greek:nTF {#4}
34510     {
34511         \exp_args:Ne \__text_change_case_breathing:nnnnn
34512         {
34513             \codepoint_to_nfd:n
34514             { \__text_codepoint_from_chars:Nw #4 }
34515         }
34516         {#1} {#2} {#3} {#4}
34517     }
34518     { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34519 }
34520 \cs_new:Npn \__text_change_case_breathing:nnnnn #1#2#3#4#5
34521 {
34522     \__text_codepoint_process:nN
34523     { \__text_change_case_breathing:nnnnnw {#2} {#3} {#4} {#5} }
34524     #1 \q_mark
34525 }

```

Normal form decomposition will always give between one and three codepoints. Luckily, the two breathing marks (*psili* and *dasia*) will be in a predictable position: last. So we

can quickly establish first that there was a change on decomposition, and second if the final resulting codepoint is one of the two we care about.

```

34526 \cs_new:Npn \__text_change_case_breathing:nnnnnw #1#2#3#4#5#6 \q_mark
34527 {
34528   \tl_if_blank:nTF {#6}
34529   { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34530   {
34531     \__text_codepoint_process:nN
34532     { \__text_change_case_breathing:nnnnnw {#1} {#2} {#3} {#4} {#5} }
34533     #6 \q_mark
34534   }
34535 }
34536 \cs_new:Npn \__text_change_case_breathing:nnnnnw #1#2#3#4#5#6#7 \q_mark
34537 {
34538   \tl_if_blank:nTF {#7}
34539   {
34540     \__text_change_case_breathing_aux:nnnnnw
34541     {#1} {#2} {#3} {#4} {#5} {#6}
34542   }
34543   {
34544     \__text_codepoint_process:nN
34545     { \__text_change_case_breathing:nnnnnw {#1} {#2} {#3} {#4} {#5} }
34546     #7 \q_mark
34547   }
34548 }
34549 \cs_new:Npn \__text_change_case_breathing_aux:nnnnnw #1#2#3#4#5#6
34550 {
34551   \bool_lazy_or:nnTF
34552   { \__text_codepoint_compare_p:nNn {#6} = { "0313 } }
34553   { \__text_codepoint_compare_p:nNn {#6} = { "0314 } }
34554   { \__text_change_case_breathing_aux:nnnw {#1} {#2} {#3} {#5} }
34555   { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34556 }

```

Now the lookahead can be fired: check the next codepoint and assess whether it takes a *dialytika*. Drop the breathing mark or generate the *dialytika*: the latter is code shared with the general mechanism.

```

34557 \cs_new:Npn \__text_change_case_breathing_aux:nnnw #1#2#3#4#5
34558   \q__text_recursion_stop
34559 {
34560   \__text_change_case_codepoint:nn { upper } {#4}
34561   \tl_if_head_is_N_type:nTF {#5}
34562   { \__text_change_case_breathing_aux:nnnN }
34563   { \__text_change_case_loop:nnnw
34564     {#1} {#2} {#3} #5 \q__text_recursion_stop
34565   }
34566 \cs_new:Npn \__text_change_case_breathing_aux:nnnN #1#2#3#4
34567 {
34568   \__text_if_q_recursion_tail_stop_do:Nn #4
34569   { \__text_change_case_break: }
34570   \__text_codepoint_process:nN
34571   { \__text_change_case_breathing_dialytika:nnnn {#1} {#2} {#3} } #4
34572 }
34573 \cs_new:Npn \__text_change_case_breathing_dialytika:nnnn #1#2#3#4

```

```

34574 {
34575     \__text_change_case_if_takes_dialytika:nTF {#4}
34576     {
34577         \__text_change_case_upper_el_dialytika:n {#4}
34578         \__text_change_case_loop:nnnw {#1} {#2} {#3}
34579     }
34580     { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
34581 }

```

(End of definition for __text_change_case_breathing:nnnn and others.)

__text_change_case_title_el:nnnnn Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

34582 \cs_new:Npn \__text_change_case_title_el:nnnnn #1#2#3#4#5
34583 { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }

```

(End of definition for __text_change_case_title_el:nnnnn.)

__text_change_case_upper_hy:nnnnn See <https://www.unicode.org/L2/L2020/20143-armenian-ech-yiwn.pdf>.

```

\__text_change_case_title_hy:nnnnn
\__text_change_case_upper_hy-x-yiwn:nnnnn
\__text_change_case_title_hy-x-yiwn:nnnnn
34584 \cs_new:Npn \__text_change_case_upper_hy:nnnnn #1#2#3#4#5
34585 {
34586     \__text_codepoint_compare:nNnTF {#5} = { "0587 }
34587     {
34588         \codepoint_generate:nn { "0535 }
34589         { \__text_change_case_catcode:nn {#5} { "0535 } }
34590         \codepoint_generate:nn { "054E }
34591         { \__text_change_case_catcode:nn {#5} { "054E } }
34592         \use:c { __text_change_case_next_ #2 :nnn }
34593         {#2} {#3} {#4}
34594     }
34595     { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34596 }
34597 \cs_new:Npn \__text_change_case_title_hy:nnnnn #1#2#3#4#5
34598 {
34599     \__text_codepoint_compare:nNnTF {#5} = { "0587 }
34600     {
34601         \codepoint_generate:nn { "0535 }
34602         { \__text_change_case_catcode:nn {#5} { "0535 } }
34603         \codepoint_generate:nn { "057E }
34604         { \__text_change_case_catcode:nn {#5} { "057E } }
34605         \use:c { __text_change_case_next_ #2 :nnn }
34606         {#2} {#3} {#4}
34607     }
34608     { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34609 }
34610 \cs_new:cpn { __text_change_case_upper_hy-x-yiwn:nnnnn } #1#2#3#4#5
34611 { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34612 \cs_new_eq:cc { __text_change_case_title_hy-x-yiwn:nnnnn }
34613 { __text_change_case_upper_hy-x-yiwn:nnnnn }

```

(End of definition for __text_change_case_upper_hy:nnnnn and others.)

__text_change_case_lower_la-x-medieval:nnnnn Simply swaps of characters.

```

\__text_change_case_upper_la-x-medieval:nnnnn
34614 \cs_new:cpn { __text_change_case_lower_la-x-medieval:nnnnn } #1#2#3#4#5
34615 {

```

```

34616     \__text_codepoint_compare:nNnTF {#5} = { "0056 }
34617     {
34618         \char_generate:nn { "0075 } { \__text_char_catcode:N #5 }
34619         \use:c { \__text_change_case_next_ #2 :nnn }
34620         {#2} {#3} {#4}
34621     }
34622     { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34623 }
34624 \cs_new:cpn { \__text_change_case_upper_la-x-medieval:nnnnn } #1#2#3#4#5
34625 {
34626     \__text_codepoint_compare:nNnTF {#5} = { "0075 }
34627     {
34628         \char_generate:nn { "0056 } { \__text_char_catcode:N #5 }
34629         \use:c { \__text_change_case_next_ #2 :nnn }
34630         {#2} {#3} {#4}
34631     }
34632     { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34633 }

```

(End of definition for __text_change_case_lower_la-x-medieval:nnnnn and __text_change_case_upper_la-x-medieval:nnnnn.)

```

\__text_change_cases_lower_lt:nnnnn
\__text_change_cases_lower_lt_auxi:nnnnn
\__text_change_cases_lower_lt_auxii:nnnnn
\__text_change_case_lower_lt:nnnw
\__text_change_case_lower_lt:nnnN
\__text_change_case_lower_lt:nnnn

```

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```

34634 \cs_new:Npn \__text_change_case_lower_lt:nnnnn #1#2#3#4#5
34635 {
34636     \exp_args:Ne \__text_change_case_lower_lt_auxi:nnnnn
34637     {
34638         \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
34639         {
34640             { "00CC } { "0300 }
34641             { "00CD } { "0301 }
34642             { "0128 } { "0303 }
34643         }
34644     }
34645     {#2} {#3} {#4} {#5}
34646 }

```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J, and I-ogonek.

```

34647 \cs_new:Npn \__text_change_case_lower_lt_auxi:nnnnn #1#2#3#4#5
34648 {
34649     \tl_if_blank:nTF {#1}
34650     {
34651         \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnnn
34652         {
34653             \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
34654             {
34655                 { "0049 } { "0069 }
34656                 { "004A } { "006A }
34657                 { "012E } { "012F }
34658             }

```

```

34659     }
34660     {#2} {#3} {#4} {#5}
34661   }
34662   {
34663     \codepoint_generate:nn { "0069 }
34664     { \_text_change_case_catcode:nn {#5} { "0069 } }
34665     \codepoint_generate:nn { "0307 }
34666     { \_text_change_case_catcode:nn {#5} { "0307 } }
34667     \codepoint_generate:nn {#1}
34668     { \_text_change_case_catcode:nn {#5} {#1} }
34669     \_text_change_case_loop:nnnw {#2} {#3} {#4}
34670   }
34671 }

```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

34672 \cs_new:Npn \_text_change_case_lower_lt_auxii:nnnnn #1#2#3#4#5
34673 {
34674   \tl_if_blank:nTF {#1}
34675   { \_text_change_case_codepoint:nnnnn {#2} {#2} {#3} {#4} {#5} }
34676   {
34677     \codepoint_generate:nn {#1}
34678     { \_text_change_case_catcode:nn {#5} {#1} }
34679     \_text_change_case_lower_lt:nnnw {#2} {#3} {#4}
34680   }
34681 }
34682 \cs_new:Npn \_text_change_case_lower_lt:nnnw #1#2#3#4 \q_text_recursion_stop
34683 {
34684   \tl_if_head_is_N_type:nTF {#4}
34685   { \_text_change_case_lower_lt:nnnN }
34686   { \_text_change_case_loop:nnnw }
34687   {#1} {#2} {#3} #4 \q_text_recursion_stop
34688 }
34689 \cs_new:Npn \_text_change_case_lower_lt:nnnN #1#2#3#4
34690 {
34691   \_text_codepoint_process:nN
34692   { \_text_change_case_lower_lt:nnnn {#1} {#2} {#3} } #4
34693 }
34694 \cs_new:Npn \_text_change_case_lower_lt:nnnn #1#2#3#4
34695 {
34696   \bool_lazy_and:nnT
34697   {
34698     \bool_lazy_or_p:nn
34699     { ! \tl_if_single_p:n {#4} }
34700     { ! \token_if_cs_p:N #4 }
34701   }
34702   {
34703     \bool_lazy_any_p:n
34704     {
34705       { \_text_codepoint_compare_p:nNn {#4} = { "0300 } }
34706       { \_text_codepoint_compare_p:nNn {#4} = { "0301 } }
34707       { \_text_codepoint_compare_p:nNn {#4} = { "0303 } }
34708     }
34709   }

```

```

34710     {
34711         \codepoint_generate:nn { "0307 }
34712         { \_text_change_case_catcode:nn {#4} { "0307 } }
34713     }
34714     \_text_change_case_loop:nnnw {#1} {#2} {#3} #4
34715 }

```

(End of definition for _text_change_cases_lower_lt:nnnnn and others.)

_text_change_cases_upper_lt:nnnnn
 _text_change_cases_upper_lt_aux:nnnnn
 _text_change_case_upper_lt:nnnw
 _text_change_case_upper_lt:nnnN
 _text_change_case_upper_lt:nnnn

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

34716 \cs_new:Npn \_text_change_case_upper_lt:nnnnn #1#2#3#4#5
34717 {
34718     \exp_args:Ne \_text_change_case_upper_lt_aux:nnnnn
34719     {
34720         \int_case:nn { \_text_codepoint_from_chars:Nw #5 }
34721         {
34722             { "0069 } { "0049 }
34723             { "006A } { "004A }
34724             { "012F } { "012E }
34725         }
34726     }
34727     {#2} {#3} {#4} {#5}
34728 }
34729 \cs_new:Npn \_text_change_case_upper_lt_aux:nnnnn #1#2#3#4#5
34730 {
34731     \tl_if_blank:nTF {#1}
34732     { \_text_change_case_codepoint:nnnnn { upper } {#2} {#3} {#4} {#5} }
34733     {
34734         \codepoint_generate:nn {#1}
34735         { \_text_change_case_catcode:nn {#5} {#1} }
34736         \_text_change_case_upper_lt:nnnw {#2} {#3} {#4}
34737     }
34738 }
34739 \cs_new:Npn \_text_change_case_upper_lt:nnnw #1#2#3#4 \q_text_recursion_stop
34740 {
34741     \tl_if_head_is_N_type:nTF {#4}
34742     { \_text_change_case_upper_lt:nnnN }
34743     { \use:c { \_text_change_case_next_ #1 :nnn } }
34744     {#1} {#2} {#3} #4 \q_text_recursion_stop
34745 }
34746 \cs_new:Npn \_text_change_case_upper_lt:nnnN #1#2#3#4
34747 {
34748     \_text_codepoint_process:nN
34749     { \_text_change_case_upper_lt:nnnn {#1} {#2} {#3} } #4
34750 }
34751 \cs_new:Npn \_text_change_case_upper_lt:nnnn #1#2#3#4
34752 {
34753     \bool_lazy_and:nnTF
34754     {
34755         \bool_lazy_or_p:nn
34756         { ! \tl_if_single_p:n {#4} }
34757         { ! \token_if_cs_p:N #4 }
34758     }

```



```

34759 { \_text_codepoint_compare_p:nNn {#4} = { "0307 } }
34760 { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} }
34761 { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
34762 }

```

(End of definition for _text_change_cases_upper_lt:nnnnn and others.)

```

\_text_change_case_title_nl:nnnnn
\_text_change_case_title_nl_aux:nnnnn
\_text_change_case_title_nl:nnnw
\_text_change_case_title_nl:nnnN

```

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

34763 \cs_new:Npn \_text_change_case_title_nl:nnnnn #1#2#3#4#5
34764 {
34765   \tl_if_single:nTF {#5}
34766   { \_text_change_case_title_nl_aux:nnnnn }
34767   { \_text_change_case_codepoint:nnnnn }
34768   {#1} {#2} {#3} {#4} {#5}
34769 }
34770 \cs_new:Npn \_text_change_case_title_nl_aux:nnnnn #1#2#3#4#5
34771 {
34772   \bool_lazy_or:nnTF
34773   { \int_compare_p:nNn {'#5} = { "0049 } }
34774   { \int_compare_p:nNn {'#5} = { "0069 } }
34775   {
34776     \char_generate:nn { "0049 } { \_text_char_catcode:N #5 }
34777     \_text_change_case_title_nl:nnnw {#2} {#3} {#4}
34778   }
34779   { \_text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34780 }
34781 \cs_new:Npn \_text_change_case_title_nl:nnnw #1#2#3#4 \q_text_recursion_stop
34782 {
34783   \tl_if_head_is_N_type:nTF {#4}
34784   { \_text_change_case_title_nl:nnnN }
34785   { \use:c { __text_change_case_next_ #1 :nnn } }
34786   {#1} {#2} {#3} #4 \q_text_recursion_stop
34787 }
34788 \cs_new:Npn \_text_change_case_title_nl:nnnN #1#2#3#4
34789 {
34790   \bool_lazy_and:nnTF
34791   { ! \token_if_cs_p:N #4 }
34792   {
34793     \bool_lazy_or_p:nn
34794     { \int_compare_p:nNn {'#4} = { "004A } }
34795     { \int_compare_p:nNn {'#4} = { "006A } }
34796   }
34797   {
34798     \char_generate:nn { "004A } { \_text_char_catcode:N #4 }
34799     \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3}
34800   }
34801   { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
34802 }

```

(End of definition for _text_change_case_title_nl:nnnnn and others.)

```

\_text_change_case_lower_tr:nnnnn
\_text_change_case_lower_tr:nnnNw
\_text_change_case_lower_tr:NnnnN
\_text_change_case_lower_tr:Nnnnn

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

34803 \cs_new:Npn \__text_change_case_lower_tr:nnnnn #1#2#3#4#5
34804 {
34805   \__text_codepoint_compare:nNnTF {#5} = { "0049 }
34806   { \__text_change_case_lower_tr:nnnNw {#1} {#3} {#4} #5 }
34807   {
34808     \__text_codepoint_compare:nNnTF {#5} = { "0130 }
34809     {
34810       \codepoint_generate:nn { "0069 }
34811       { \__text_change_case_catcode:nn {#5} { "0069 } }
34812       \__text_change_case_loop:nnnw {#1} {#3} {#4}
34813     }
34814     { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34815   }
34816 }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```

34817 \cs_new:Npn \__text_change_case_lower_tr:nnnNw #1#2#3#4#5 \q__text_recursion_stop
34818 {
34819   \tl_if_head_is_N_type:nTF {#5}
34820   { \__text_change_case_lower_tr:NnnnN #4 {#1} {#2} {#3} }
34821   {
34822     \codepoint_generate:nn { "0131 }
34823     { \__text_change_case_catcode:nn {#4} { "0131 } }
34824     \__text_change_case_loop:nnnw {#1} {#2} {#3}
34825   }
34826   #5 \q__text_recursion_stop
34827 }
34828 \cs_new:Npn \__text_change_case_lower_tr:NnnnN #1#2#3#4#5
34829 {
34830   \__text_codepoint_process:nN
34831   { \__text_change_case_lower_tr:Nnnnn #1 {#2} {#3} {#4} } #5
34832 }
34833 \cs_new:Npn \__text_change_case_lower_tr:Nnnnn #1#2#3#4#5
34834 {
34835   \bool_lazy_or:nnTF
34836   {
34837     \bool_lazy_and_p:nn
34838     { \tl_if_single_p:n {#5} }
34839     { \token_if_cs_p:N #5 }
34840   }
34841   { ! \__text_codepoint_compare_p:nNn {#5} = { "0307 } }
34842   {
34843     \codepoint_generate:nn { "0131 }
34844     { \__text_change_case_catcode:nn {#1} { "0131 } }
34845     \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
34846   }
34847   {
34848     \codepoint_generate:nn { "0069 }
34849     { \__text_change_case_catcode:nn {#1} { "0069 } }
34850     \__text_change_case_loop:nnnw {#2} {#3} {#4}
34851   }
34852 }

```

(End of definition for `\_text\_change\_case\_lower\_tr:nnnnn` and others.)

`\_text\_change\_case\_upper\_tr:nnnnn`

Uppercasing is easier: just one exception with no context.

```

34853 \cs_new:Npn \_text\_change\_case\_upper\_tr:nnnnn #1#2#3#4#5
34854 {
34855   \_text\_codepoint\_compare:nNnTF {#5} = { "0069 }
34856   {
34857     \codepoint\_generate:nn { "0130 }
34858     { \_text\_change\_case\_catcode:nn {#5} { "0130 } }
34859     \use:c { \_text\_change\_case\_next\_ #2 :nnn } {#2} {#3} {#4}
34860   }
34861   { \_text\_change\_case\_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
34862 }

```

(End of definition for `\_text\_change\_case\_upper\_tr:nnnnn`.)

`\_text\_change\_case\_lower\_az:nnnnn`

Straight copies.

`\_text\_change\_case\_upper\_az:nnnnn`

```

34863 \cs_new_eq:NN \_text\_change\_case\_lower\_az:nnnnn
34864   \_text\_change\_case\_lower\_tr:nnnnn
34865 \cs_new_eq:NN \_text\_change\_case\_upper\_az:nnnnn
34866   \_text\_change\_case\_upper\_tr:nnnnn

```

(End of definition for `\_text\_change\_case\_lower\_az:nnnnn` and `\_text\_change\_case\_upper\_az:nnnnn`.)

The (fixed) look-up mappings for letter-like control sequences.

```

34867 \group\_begin:
34868   \cs\_set\_protected:Npn \_text\_change\_case\_setup:NN #1#2
34869   {
34870     \quark\_if\_recursion\_tail\_stop:N #1
34871     \tl\_const:cn { c\_text\_lowercase\_ \token\_to\_str:N #1 \_tl }
34872     { #2 }
34873     \tl\_const:cn { c\_text\_uppercase\_ \token\_to\_str:N #2 \_tl }
34874     { #1 }
34875     \_text\_change\_case\_setup:NN
34876   }
34877   \_text\_change\_case\_setup:NN
34878   \AA \aa
34879   \AE \ae
34880   \DH \dh
34881   \DJ \dj
34882   \IJ \ij
34883   \L \l
34884   \NG \ng
34885   \O \o
34886   \OE \oe
34887   \SS \ss
34888   \TH \th
34889   \q\_recursion\_tail ?
34890   \q\_recursion\_stop
34891   \tl\_const:cn { c\_text\_uppercase\_ \token\_to\_str:N \i \_tl } { I }
34892   \tl\_const:cn { c\_text\_uppercase\_ \token\_to\_str:N \j \_tl } { J }
34893 \group\_end:

```

To deal with possible encoding-specific extensions to `\@uclclist`, we check at the end of the preamble. This will therefore only apply to L^AT_EX 2_ε package mode.

```

34894 \tl_if_exist:NT \@expl@finalise@setup@@
34895 {
34896   \tl_gput_right:Nn \@expl@finalise@setup@@
34897   {
34898     \tl_gput_right:Nn \@kernel@after@begindocument
34899     {
34900       \group_begin:
34901       \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
34902       {
34903         \quark_if_recursion_tail_stop:N #1
34904         \tl_if_single_token:nT {#2}
34905         {
34906           \cs_if_exist:cF
34907           { c__text_uppercase_ \token_to_str:N #1 _tl }
34908           {
34909             \tl_const:cn
34910             { c__text_uppercase_ \token_to_str:N #1 _tl }
34911             { #2 }
34912           }
34913           \cs_if_exist:cF
34914           { c__text_lowercase_ \token_to_str:N #2 _tl }
34915           {
34916             \tl_const:cn
34917             { c__text_lowercase_ \token_to_str:N #2 _tl }
34918             { #1 }
34919           }
34920         }
34921         \__text_change_case_setup:Nn
34922       }
34923       \exp_after:wN \__text_change_case_setup:Nn \@uclclist
34924       \q_recursion_tail ?
34925       \q_recursion_stop
34926       \group_end:
34927     }
34928   }
34929 }

```

A few adjustments to case mapping for combining chars: these are not needed for the Unicode engines

```

34930 \sys_if_engine_opentype:F
34931 {
34932   \text_declare_uppercase_mapping:nn { "01F0 } { \v { J } }
34933 }
34934 </code>

```

Chapter 94

text-map implementation

```
34935 (*code)
34936 (@@=text)
```

94.1 Mapping to text

94.1.1 Common code

Mapping to text all works the same way: using standard “action” loop on expanded text. There are different ways to determine the boundary conditions for breaking: to avoid duplication, the common ideas are covered here with the specifics split out. In all cases, anything which is not a character token is treated as a boundary.

```

__text_map_tokens:nnn
__text_map_tokens:enn
  __text_map_loop:nnnw
  __text_map_group:nnnn
  __text_map_space:nnnw
  __text_map_N_type:nnnN
    __text_map_math_search:nnnnN
    __text_map_math_search:nnnNNN
__text_map_math_loop:nnnNw
  __text_map_math_N_type:nnnNN
__text_map_math_group:nnnNn
__text_map_math_space:nnnNw
  __text_map_cs_check:nnnN
    __text_map_active_check:nnnn
__text_map_codepoint:nnnn
  __text_map_CR:nnnw
  __text_map_CR:nnnN
  __text_map_class:nnnn
  __text_map_class:nnnnn
__text_map_lookahead:nnnnnw
__text_map_lookahead:nnnnnN
__text_map_if_ignorable:nTF
  __text_map_output:nn
    \text_map_break:
    \text_map_break:n
34937 \cs_new:Npn \__text_map_tokens:nnn #1#2#3
34938 {
34939   __text_map_loop:nnnw {#3} {#2} { } #1
34940   \q__text_recursion_tail \q__text_recursion_stop
34941   \prg_break_point:Nn \text_map_break: { }
34942 }
34943 \cs_generate_variant:Nn \__text_map_tokens:nnn { e }
34944 \cs_new:Npn \__text_map_loop:nnnw #1#2#3#4 \q__text_recursion_stop
34945 {
34946   \tl_if_head_is:N_type:nTF {#4}
34947   { \__text_map_N_type:nnnN }
34948   {
34949     \tl_if_head_is_group:nTF {#4}
34950     { \__text_map_group:nnnn }
34951     { \__text_map_space:nnnw }
34952   }
34953   {#1} {#2} {#3} #4 \q__text_recursion_stop
34954 }
34955 \cs_new:Npn \__text_map_group:nnnn #1#2#3#4
34956 {
34957   __text_map_output:nn {#1} {#3}
34958   __text_map_output:nn {#1} { {#4} }
34959   __text_map_loop:nnnw {#1} {#2} { }
34960 }
34961 \use:e
34962 { \cs_new:Npn \exp_not:N \__text_map_space:nnnw #1#2#3 \c_space_tl }
```

```

34963 {
34964   \_text_map_output:nn {#1} {#3}
34965   #1 { ~ }
34966   \_text_map_loop:nnnw {#1} {#2} { }
34967 }
34968 \cs_new:Npn \_text_map_N_type:nnnN #1#2#3#4
34969 {
34970   \_text_if_q_recursion_tail_stop_do:Nn #4
34971   {
34972     \_text_map_output:nn {#1} {#3}
34973     \text_map_break:
34974   }
34975   \exp_args:NV \_text_map_math_search:nnnnN
34976   \l_text_math_delims_tl {#1} {#2} {#3} #4
34977 }

```

We need to exclude math mode here as quite apart from the conceptual questions, it could contain implicit tokens. Those would cause all sorts of issues, so are best just skipped over here.

```

34978 \cs_new:Npn \_text_map_math_search:nnnnN #1#2#3#4#5
34979 {
34980   \_text_map_math_search:nnnNNN {#2} {#3} {#4} #5 #1
34981   \q_text_recursion_tail \q_text_recursion_tail
34982   \q_text_recursion_stop
34983 }
34984 \cs_new:Npn \_text_map_math_search:nnnNNN #1#2#3#4#5#6
34985 {
34986   \_text_if_q_recursion_tail_stop_do:Nn #5
34987   { \_text_map_cs_check:nnnN {#1} {#2} {#3} #4 }
34988   \token_if_eq_meaning:NNTF #4 #5
34989   {
34990     \_text_use_i_delimit_by_q_recursion_stop:nw
34991     {
34992       \_text_map_output:nn {#1} {#3}
34993       \_text_map_math_loop:nnnNw {#1} {#2} {#4} #6
34994     }
34995   }
34996   { \_text_map_math_search:nnnNNN {#1} {#2} {#3} #4 }
34997 }
34998 \cs_new:Npn \_text_map_math_loop:nnnNw #1#2#3#4#5 \q_text_recursion_stop
34999 {
35000   \tl_if_head_is_N_type:nTF {#5}
35001   { \_text_map_math_N_type:nnnNN }
35002   {
35003     \tl_if_head_is_group:nTF {#5}
35004     { \_text_map_math_group:nnnNn }
35005     { \_text_map_math_space:nnnNw }
35006   }
35007   {#1} {#2} {#3} #4 #5 \q_text_recursion_stop
35008 }
35009 \cs_new:Npn \_text_map_math_N_type:nnnNN #1#2#3#4#5
35010 {
35011   \_text_if_q_recursion_tail_stop_do:Nn #5
35012   {

```

```

35013     \_text_map_output:nn {#1} {#3}
35014     \text_map_break:
35015   }
35016   \token_if_eq_meaning:NNTF #5 #4
35017   {
35018     \_text_map_output:nn {#1} { #3 #4 }
35019     \_text_map_loop:nnnw {#1} {#2} { }
35020   }
35021   { \_text_map_math_loop:nnnNw {#1} {#2} { #3 #5 } #4 }
35022 }
35023 \cs_new:Npn \_text_map_math_group:nnnNn #1#2#3#4#5
35024 { \_text_map_math_loop:nnnNw {#1} {#2} { #3 {#5} } #4 }
35025 \use:e
35026 {
35027   \cs_new:Npn \exp_not:N \_text_map_math_space:nnnNw #1#2#3#4
35028   \c_space_tl
35029 }
35030 { \_text_map_math_loop:nnnNw {#1} {#2} { #3 ~ } #4 }
35031 \cs_new:Npn \_text_map_cs_check:nnnN #1#2#3#4
35032 {
35033   \token_if_cs:NNTF #4
35034   {
35035     \_text_map_output:nn {#1} {#3}
35036     #1 {#4}
35037     \_text_map_loop:nnnw {#1} {#2} { }
35038   }
35039   {
35040     \_text_codepoint_process:nN
35041     { \_text_map_active_check:nnnn {#1} {#2} {#3} } #4
35042   }
35043 }
35044 \cs_new:Npn \_text_map_active_check:nnnn #1#2#3#4
35045 {
35046   \bool_lazy_and:nnTF
35047   { \tl_if_single_token_p:n {#4} }
35048   { \token_if_active_p:N #4 }
35049   {
35050     \_text_map_output:nn {#1} {#3}
35051     #1 {#4}
35052     \_text_map_loop:nnnw {#1} {#2} { }
35053   }
35054   { \_text_map_codepoint:nnnn {#1} {#2} {#3} {#4} }
35055 }

```

We pull out a few special cases here. Carriage returns case needs a bit of context handling so has an auxiliary. Codepoint U+200D is the zero-width joiner, which has no context to concern us: just don't break. (These special cases apply to all forms of text mapping.)

```

35056 \cs_new:Npn \_text_map_codepoint:nnnn #1#2#3#4
35057 {
35058   \_text_codepoint_compare:nNnTF {#4} = { "000D }
35059   {
35060     \_text_map_output:nn {#1} {#3}
35061     \_text_map_CR:nnnw {#1} {#2} {#4}
35062   }

```

```

35063     {
35064         \__text_codepoint_compare:nNnTF {#4} = { "200D }
35065         { \__text_map_loop:nnnw {#1} {#2} {#3#4} }
35066         { \__text_map_class:nnnn {#1} {#2} {#3} {#4} }
35067     }
35068 }

```

A carriage return is a boundary unless it is immediately followed by a line feed, in which case that pair is a boundary.

```

35069 \cs_new:Npn \__text_map_CR:nnnw #1#2#3#4 \q__text_recursion_stop
35070 {
35071     \tl_if_head_is_N_type:nTF {#4}
35072     { \__text_map_CR:nnnN {#1} {#2} {#3} }
35073     {
35074         #1 {#3}
35075         \__text_map_loop:nnnw {#1} {#2} { }
35076     }
35077     #4 \q__text_recursion_stop
35078 }
35079 \cs_new:Npn \__text_map_CR:nnnN #1#2#3#4
35080 {
35081     \__text_if_q_recursion_tail_stop_do:Nn #4
35082     {
35083         #1 {#3}
35084         \text_map_break:
35085     }
35086     \bool_lazy_and:nnTF
35087     { ! \token_if_cs_p:N #4 }
35088     { \int_compare_p:nNn { '#4 } = { "000A } }
35089     {
35090         \__text_map_output:nn {#1} {#3#4}
35091         \__text_map_loop:nnnw {#1} {#2} { }
35092     }
35093     { \__text_map_loop:nnnw {#1} {#2} { } #3 }
35094 }

```

There are various classes of character, and we deal with them all in the same general way. We need to example the relevant list of codepoints: if we get a hit, then we do whatever the relevant action is. To keep names short and to allow code sharing, we have two ways of naming the functions: most class names are unique, so it's only where we see the same name used in both break classes that we need more control.

```

35095 \cs_new:Npn \__text_map_class:nnnn #1#2#3#4
35096 {
35097     \exp_args:Nnnne \__text_map_class:nnnnn {#1} {#2} {#3} {#4}
35098     {
35099         \use:c { __kernel_codepoint_to_ #2 _class:n }
35100         { \__text_codepoint_from_chars:Nw #4 }
35101     }
35102 }
35103 \cs_new:Npn \__text_map_class:nnnnn #1#2#3#4#5
35104 {
35105     \cs_if_exist_use:cF { __text_map_ #5 :nnnn }
35106     { \__text_map_Other:nnnn }
35107     {#1} {#2} {#3} {#4}
35108 }

```


A generic loop-ahead setup: we need to handle both the previously collected tokens and any “conditional” ones. The latter occur when looking ahead for word-breaking: these *may* be combined with the collected tokens, but if we hit the end-of-loop, need to be output separately.

```

35109 \cs_new:Npn \__text_map_lookahead:nnnnnw #1#2#3#4#5#6 \q__text_recursion_stop
35110 {
35111   \tl_if_head_is_N_type:nTF {#6}
35112     { \__text_map_lookahead:nnnnnN {#1} {#2} {#3} {#4} {#5} }
35113     { \__text_map_loop:nnnw {#1} {#2} {#3} #4 }
35114   #6 \q__text_recursion_stop
35115 }
35116 \cs_new:Npn \__text_map_lookahead:nnnnnN #1#2#3#4#5#6
35117 {
35118   \__text_if_q_recursion_tail_stop_do:Nn #6
35119   {
35120     #1 {#3}
35121     \tl_if_blank:nF {#4} { #1 {#4} }
35122   }
35123   \token_if_cs:NTF #6
35124   {
35125     #1 {#3}
35126     \__text_map_loop:nnnw {#1} {#2} { } #4
35127   }
35128   { \__text_codepoint_process:nN { #5 {#1} {#2} {#3} {#4} } }
35129   #6
35130 }

```

To deal with “ignored” characters for word break mapping: needed for generic `Regional_Indicator` function, so set up here.

```

35131 \prg_new_conditional:Npnn \__text_map_if_ignorable:n #1 { TF }
35132 {
35133   \str_case:nnTF {#1}
35134   {
35135     { Extend }      { }
35136     { Format }      { }
35137     { ZWJ }        { }
35138   }
35139   \prg_return_true:
35140   \prg_return_false:
35141 }

```

For the end of the process.

```

35142 \cs_new:Npn \__text_map_output:nn #1#2
35143 { \tl_if_blank:nF {#2} { #1 {#2} } }
35144 \cs_new:Npn \text_map_break:
35145 { \prg_map_break:Nn \text_map_break: { } }
35146 \cs_new:Npn \text_map_break:n
35147 { \prg_map_break:Nn \text_map_break: }

```

(End of definition for __text_map_tokens:nnn and others. These functions are documented on page 308.)

```

\__text_map_Control:nnnn
\__text_map_Newline:nnnn
\__text_map_Extend:nnnn
\__text_map_Format:nnnn
\__text_map_SpacingMark:nnnn
\__text_map_Other:nnnn
\__text_map_Regional_Indicator:nnnn
\__text_map_Regional_Indicator_aux:nnnnn

```

A small number of classes appear in both forms of breaking and have the same behavior. For **Control** and **Newline**, we set up here as they are the same outcome. We have the same story for **Format**, which is functionally the same as **Newline**.

```

35148 \cs_new:Npn \__text_map_Control:nnnn #1#2#3#4
35149 {
35150   \__text_map_output:nn {#1} {#3}
35151   \__text_map_output:nn {#1} {#4}
35152   \__text_map_loop:nnnw {#1} {#2} { }
35153 }
35154 \cs_new_eq:NN \__text_map_Newline:nnnn \__text_map_Control:nnnn
35155 \cs_new:Npn \__text_map_Extend:nnnn #1#2#3#4
35156 { \__text_map_loop:nnnw {#1} {#2} {#3#4} }
35157 \cs_new_eq:NN \__text_map_Format:nnnn \__text_map_Extend:nnnn
35158 \cs_new_eq:NN \__text_map_SpacingMark:nnnn \__text_map_Extend:nnnn
35159 \cs_new:Npn \__text_map_Other:nnnn #1#2#3#4
35160 {
35161   \__text_map_output:nn {#1} {#3}
35162   \__text_map_loop:nnnw {#1} {#2} {#4}
35163 }

```

The Regional Indicator rule means looking ahead and dealing with the case where there are two in a row. So we use a look ahead to pick them off. As there is only one range the values are hard-coded. For word breaking, we also need to allow for the various extenders.

```

35164 \cs_new:Npn \__text_map_Regional_Indicator:nnnn #1#2#3#4
35165 {
35166   \__text_map_output:nn {#1} {#3}
35167   \__text_map_lookahead:nnnnnw {#1} {#2} {#4} { }
35168   \__text_map_Regional_Indicator_aux:nnnnn
35169 }
35170 \cs_new:Npn \__text_map_Regional_Indicator_aux:nnnnn #1#2#3#4#5
35171 {
35172   \bool_lazy_or:nnTF
35173   { \__text_codepoint_compare_p:nNn {#5} < { "1F1E6 } }
35174   { \__text_codepoint_compare_p:nNn {#5} > { "1F1FF } }
35175   {
35176     \str_if_eq:nnTF {#2} { wordbreak }
35177     {
35178       \exp_args:Ne \__text_map_if_ignorable:nTF
35179       {
35180         \__kernel_codepoint_to_grapheme_class:n
35181         { \__text_codepoint_from_chars:Nw #5 }
35182       }
35183       {
35184         \__text_map_lookahead:nnnnnw {#1} {#2} {#3#5} { }
35185         \__text_map_Regional_Indicator_aux:nnnnn
35186       }
35187       { \__text_map_loop:nnnw {#1} {#2} {#3} #5 }
35188     }
35189     { \__text_map_loop:nnnw {#1} {#2} {#3} #5 }
35190   }
35191   { \__text_map_loop:nnnw {#1} {#2} {#3#5} }
35192 }

```

(End of definition for __text_map_Control:nnnn and others.)

94.2 Grapheme mapping

\text_map_function:nN

\text_map_tokens:nn

__text_map_Prepend:nnnn

__text_map_Prepend_aux:nnnnn

__text_map_Prepend:nnn

__text_map_L:nnnn

__text_map_LV:nnnn

__text_map_V:nnnn

__text_map_LVT:nnnn

__text_map_T:nnnn

__text_map_hangul:nnnw

__text_map_hangul:nnnN

__text_map_hangul:nnnn

__text_map_hangul_aux:nnnnw

__text_map_hangul:nnnnw

__text_map_hangul_next:nnnnn

__text_map_hangul_end:nw

__text_map_hangul_L:nnn

__text_map_hangul_LV:nnn

__text_map_hangul_V:nnn

__text_map_hangul_LVT:nnn

__text_map_hangul_T:nnn

The standard lead-off for an action loop.

```

35193 \cs_new:Npn \text_map_function:nN #1#2
35194 {
35195   \__text_map_tokens:enn { \text_expand:n {#1} }
35196   { grapheme } {#2}
35197 }
35198 \cs_new:Npn \text_map_tokens:nn #1#2
35199 {
35200   \__text_map_tokens:enn { \text_expand:n {#1} }
35201   { grapheme } {#2}
35202 }
```

Outputting anything earlier, the combine with what follows. The only exclusions are control characters.

```

35203 \cs_new:Npn \__text_map_Prepend:nnnn #1#2#3#4
35204 {
35205   \__text_map_output:nn {#1} {#3}
35206   \__text_map_lookahead:nnnnnw {#1} { grapheme } {#4} { }
35207   \__text_map_Prepend_aux:nnnnn
35208 }
35209 \cs_new:Npn \__text_map_Prepend_aux:nnnnn #1#2#3#4#5
35210 {
35211   \bool_lazy_or:nnTF
35212   { \__text_codepoint_compare_p:nNn {#5} = { "000A } }
35213   { \__text_codepoint_compare_p:nNn {#5} = { "000D } }
35214   {
35215     #1 {#3}
35216     \__text_map_loop:nnnw {#1} { grapheme } {#5}
35217   }
35218   { \__text_map_Prepend:nnn {#1} {#3} {#5} }
35219 }
35220 \cs_new:Npn \__text_map_Prepend:nnn #1#2#3
35221 {
35222   \str_if_eq:eeTF
35223   { Control }
35224   {
35225     \__kernel_codepoint_to_grapheme_class:n
35226     { \__text_codepoint_from_chars:Nw #3 }
35227   }
35228   { \__text_map_loop:nnnw {#1} { grapheme } {#2} #3 }
35229   { \__text_map_loop:nnnw {#1} { grapheme } {#2#3} }
35230 }
```

Hangul needs additional treatment. First we have to deal with the start-of-Hangul position: output what we had up to now, then move the specialist handler. The idea here is to pick off the different codepoint types one at a time, tracking what else can be considered at each stage until we hit the end of the viable types. Other than that, we just keep building up the Hangul codepoints using a dedicated version of the loop from above.

```

35231 \cs_new:Npn \__text_map_L:nnnn #1#2#3#4
35232 {
35233   \__text_map_output:nn {#1} {#3}
35234   \__text_map_hangul:nnnw
```

```

35235     {#1} {#4} { L ; V ; LV ; LVT }
35236 }
35237 \cs_new:Npn \__text_map_LV:nnnn #1#2#3#4
35238 {
35239     \__text_map_output:nn {#1} {#3}
35240     \__text_map_hangul:nnnw
35241     {#1} {#4} { V ; T }
35242 }
35243 \cs_new_eq:NN \__text_map_V:nnnn \__text_map_LV:nnnn
35244 \cs_new:Npn \__text_map_LVT:nnnn #1#2#3#4
35245 {
35246     \__text_map_output:nn {#1} {#3}
35247     \__text_map_hangul:nnnw
35248     {#1} {#4} { T }
35249 }
35250 \cs_new_eq:NN \__text_map_T:nnnn \__text_map_LVT:nnnn
35251 \cs_new:Npn \__text_map_hangul:nnnw #1#2#3#4 \q_text_recursion_stop
35252 {
35253     \tl_if_head_is_N_type:nTF {#4}
35254     { \__text_map_hangul:nnnN {#1} {#2} {#3} }
35255     {
35256         #1 {#2}
35257         \__text_map_loop:nnnw {#1} { grapheme } { }
35258     }
35259     #4 \q_text_recursion_stop
35260 }
35261 \cs_new:Npn \__text_map_hangul:nnnN #1#2#3#4
35262 {
35263     \__text_if_q_recursion_tail_stop_do:Nn #4
35264     {
35265         #1 {#2}
35266         \text_map_break:
35267     }
35268     \token_if_cs:NTF #4
35269     {
35270         #1 {#2}
35271         \__text_map_loop:nnnw {#1} { grapheme } { }
35272     }
35273     {
35274         \__text_codepoint_process:nN
35275         { \__text_map_hangul:nnnn {#1} {#2} {#3} } #4
35276     }
35277 }
35278 \exp_args_generate:n { Nnne }
35279 \cs_new:Npn \__text_map_hangul:nnnn #1#2#3#4
35280 {
35281     \exp_args:Nnne \__text_map_hangul_aux:nnnnw {#1} {#2} {#4}
35282     {
35283         \__kernel_codepoint_to_grapheme_class:n
35284         { \__text_codepoint_from_chars:Nw #4 }
35285     }
35286     #3 ; \q_recursion_tail ; \q_recursion_stop
35287 }
35288 \cs_new:Npn \__text_map_hangul_aux:nnnnw #1#2#3#4#5 ;

```

```

35289 {
35290   \quark_if_recursion_tail_stop_do:nn {#5}
35291   { \__text_map_loop:nnnw {#1} { grapheme } {#2} #3 }
35292   \__text_map_hangul:nnnnnw {#1} {#2} {#3} {#4} {#5}
35293 }
35294 \cs_generate_variant:Nn \__text_map_hangul_aux:nnnw { nnne }
35295 \cs_new:Npn \__text_map_hangul:nnnnnw #1#2#3#4#5#6 \q_recursion_stop
35296 {
35297   \str_if_eq:nnTF {#4} {#5}
35298   { \use:c { \__text_map_hangul_ #5 :nnn } {#1} {#2} {#3} }
35299   { \__text_map_hangul_next:nnnnn {#1} {#2} {#3} {#4} {#6} }
35300 }
35301 \cs_new:Npn \__text_map_hangul_next:nnnnn #1#2#3#4#5
35302 { \__text_map_hangul_aux:nnnw {#1} {#2} {#3} {#4} #5 \q_recursion_stop }
35303 \cs_new:Npn \__text_map_hangul_end:nw #1#2 \q__text_recursion_stop {#1}
35304 \cs_new:Npn \__text_map_hangul_L:nnn #1#2#3
35305 {
35306   \__text_map_hangul:nnnw
35307   {#1} {#2#3} { L V { LV } { LVT } }
35308 }
35309 \cs_new:Npn \__text_map_hangul_LV:nnn #1#2#3
35310 {
35311   \__text_map_hangul:nnnw
35312   {#1} {#2#3} { VT }
35313 }
35314 \cs_new_eq:NN \__text_map_hangul_V:nnn \__text_map_hangul_LV:nnn
35315 \cs_new:Npn \__text_map_hangul_LVT:nnn #1#2#3
35316 {
35317   \__text_map_hangul:nnnw
35318   {#1} {#2#3} { T }
35319 }
35320 \cs_new_eq:NN \__text_map_hangul_T:nnn \__text_map_hangul_LVT:nnn

```

(End of definition for `\text_map_function:nN` and others. These functions are documented on page 307.)

94.3 Word break mapping

`\text_map_function:nN`

The standard lead-off for an action loop.

`\text_map_tokens:nn`

```

35321 \cs_new:Npn \text_words_map_function:nN #1#2
35322 {
35323   \__text_map_tokens:enn { \text_expand:n {#1} }
35324   { wordbreak } {#2}
35325 }
35326 \cs_new:Npn \text_words_map_tokens:nn #1#2
35327 {
35328   \__text_map_tokens:enn { \text_expand:n {#1} }
35329   { wordbreak } {#2}
35330 }

```

The main rule for word breaking is that characters bind to following ones, potentially either allowing for *or* totally ignoring intervening ones. For each class, we are passed a list of classes that bind and ones that we should allow in between. In all cases, the classes **Extend**, **Format** and **ZWJ** need to be entirely ignored: they are hard coded and handled

separately from the in-between ones. Notice that we use `\str_case:nnTF` to make our boolean here: that way, all that needs to be passed internally are lists of classes.

```

35331 \cs_new:Npn \__text_map_collect:nnnnn #1#2#3#4#5
35332 {
35333   \__text_map_lookahead:nnnnnw {#1} { wordbreak } {#2} { }
35334   { \__text_map_collect_auxi:nnnnnnnn {#3} {#4} {#5} }
35335 }
35336 \cs_new:Npn \__text_map_collect_auxi:nnnnnnnn #1#2#3#4#5#6#7#8
35337 {
35338   \exp_args:Ne \__text_map_collect_auxii:nnnnnnnn
35339   {
35340     \kernel_codepoint_to_wordbreak_class:n
35341     { \__text_codepoint_from_chars:Nw #8 }
35342   }
35343   {#4} {#6} {#1} {#2} {#3} {#8}
35344 }

```

We now need to deal with the three possible positive outcomes of examining the next character. The first is that we have found one of the binding characters that ends the current cycle: we then pass on to the appropriate function. Second, we have the ignored characters: if we find these, we loop back around. Finally, we look at the “in-between” characters: if one is found, we need a further look ahead to reach a decision. Rather than have extra complexity in the setup, we have a hard-coded skipping of `ExtendNumLet` for `WSegSpace` (as `ExtendNumLet` only applies to `ALetter`, `Hebrew_Letter`, `Numeric` and `Katakana`).

```

35345 \cs_new:Npn \__text_map_collect_auxii:nnnnnnnn #1#2#3#4#5#6#7
35346 {
35347   \str_case:neTF {#1}
35348   {
35349     \tl_map_function:eN
35350     {
35351       #4
35352       \str_if_eq:nnF {#4} { { WSegSpace } } { { ExtendNumLet } }
35353     }
35354     \__text_map_collect_auxiii:n
35355   }
35356   {
35357     \cs_if_exist_use:cF { __text_map_ #1 :nnnn }
35358     { \__text_map_Other:nnnn }
35359     {#2} { wordbreak } { } {#3#7}
35360   }
35361   {
35362     \__text_map_if_ignorable:nTF {#1}
35363     { \__text_map_collect:nnnnn {#2} {#3#7} {#4} {#5} {#6} }
35364     {
35365       \str_case:neTF {#1}
35366       { \tl_map_function:nN {#5} \__text_map_collect_auxiii:n }
35367       {
35368         \__text_map_lookahead:nnnnnw {#2} { wordbreak } {#3} {#7}
35369         { \__text_map_collect_auxiv:nnnnnnnn {#5} {#6} }
35370       }
35371       {
35372         \__text_map_output:nn {#2} {#3}
35373         \__text_map_loop:nnnw {#2} { wordbreak } { } #7

```

```

35374         }
35375     }
35376 }
35377 }
35378 \cs_new:Npn \__text_map_collect_auxiii:n #1
35379 { \exp_not:n { {#1} { } } }

```

We are now have a character which *may* bind to the previous one if the next character is of the correct class also. So we carry forward the collected material and the conditional character, then look ahead again. If successful, combine together and move on using the new class, otherwise output and restart where we were.

```

35380 \cs_new:Npn \__text_map_collect_auxiv:nnnnnnn #1#2#3#4#5#6#7
35381 {
35382     \exp_args:Ne \__text_map_collect_auxv:nnnnnnn
35383     {
35384         \__kernel_codepoint_to_wordbreak_class:n
35385         { \__text_codepoint_from_chars:Nw #7 }
35386     }
35387     {#3} {#5} {#6} {#1} {#2} {#7}
35388 }
35389 \cs_new:Npn \__text_map_collect_auxv:nnnnnnn #1#2#3#4#5#6#7
35390 {
35391     \str_case:neTF {#1}
35392     { \tl_map_function:nN {#6} \__text_map_collect_auxiii:n }
35393     { \use:c { \__text_map_ #1 :nnnn } {#2} { wordbreak } { } {#3#4#7} }
35394     {
35395         \__text_map_if_ignorable:nTF {#1}
35396         {
35397             \__text_map_lookahead:nnnnnw {#2} { wordbreak } {#3} {#4#7}
35398             { \__text_map_collect_auxiv:nnnnnnn {#5} {#6} }
35399         }
35400         {
35401             \__text_map_output:nn {#2} {#3}
35402             \__text_map_loop:nnnw {#2} { wordbreak } { } {#4#7}
35403         }
35404     }
35405 }

```

Use the generic collector.

```

35406 \cs_new:Npn \__text_map_ALetter:nnnn #1#2#3#4
35407 {
35408     \__text_map_output:nn {#1} {#3}
35409     \__text_map_collect:nnnnn {#1} {#4}
35410     { { ALetter } { Hebrew_Letter } { Numeric } }
35411     { { MidLetter } { MidNumLet } { Single_Quote } }
35412     { { ALetter } { Hebrew_Letter } }
35413 }
35414 \cs_new:Npn \__text_map_Hebrew_Letter:nnnn #1#2#3#4
35415 {
35416     \__text_map_output:nn {#1} {#3}
35417     \__text_map_collect:nnnnn {#1} {#4}
35418     { { ALetter } { Hebrew_Letter } { Numeric } { Single_Quote } }
35419     { { MidLetter } { MidNumLet } { Double_Quote } }
35420     { { Hebrew_Letter } }
35421 }

```

```

35422 \cs_new:Npn \__text_map_Katakana:nnnn #1#2#3#4
35423 {
35424   \__text_map_output:nn {#1} {#3}
35425   \__text_map_collect:nnnn {#1} {#4} { { Katakana } } { } { }
35426 }
35427 \cs_new:Npn \__text_map_Numeric:nnnn #1#2#3#4
35428 {
35429   \__text_map_output:nn {#1} {#3}
35430   \__text_map_collect:nnnn {#1} {#4}
35431   { { ALetter } { Hebrew_Letter } { Numeric } }
35432   { { MidNum } { MidNumLet } { Single_Quote } }
35433   { { Numeric } }
35434 }
35435 \cs_new:Npn \__text_map_WSegSpace:nnnn #1#2#3#4
35436 {
35437   \__text_map_output:nn {#1} {#3}
35438   \__text_map_collect:nnnn {#1} {#4} { { WSegSpace } } { } { }
35439 }

```

We should only get here in the case we have a “dangling” extender. If so, look ahead for characters to bind to, then for the set of three that we need to skip over.

```

35440 \cs_new:Npn \__text_map_ExtendNumLet:nnnn #1#2#3#4
35441 {
35442   \__text_map_output:nn {#1} {#3}
35443   \__text_map_lookahead:nnnnw {#1} { wordbreak } {#4} { }
35444   \__text_map_ExtendNumLet_auxi:nnnn
35445 }
35446 \cs_new:Npn \__text_map_ExtendNumLet_auxi:nnnn #1#2#3#4#5
35447 {
35448   \exp_args:Ne \__text_map_ExtendNumLet_auxii:nnnn
35449   {
35450     \__kernel_codepoint_to_wordbreak_class:n
35451     { \__text_codepoint_from_chars:Nw #5 }
35452   }
35453   {#1} {#3} {#5}
35454 }
35455 \cs_new:Npn \__text_map_ExtendNumLet_auxii:nnnn #1#2#3#4
35456 {
35457   \str_case:nnTF {#1}
35458   {
35459     { ALetter }      { }
35460     { Hebrew_Letter } { }
35461     { Numeric }      { }
35462     { Katakana }     { }
35463     { ExtendNumLet } { }
35464   }
35465   {
35466     \cs_if_exist_use:cF { __text_map_ #1 :nnnn }
35467     { \__text_map_Other:nnnn }
35468     {#2} { wordbreak } { } {#3#4}
35469   }
35470   {
35471     \__text_map_if_ignorable:nTF {#1}
35472     {

```



```

35473         \__text_map_lookahead:nnnnnw {#2} { wordbreak } {#3#4} { }
35474         \__text_map_ExtendNumLet_auxi:nnnnn
35475     }
35476     {
35477         \__text_map_output:nn {#2} {#3}
35478         \__text_map_loop:nnnw {#2} { wordbreak } { } #4
35479     }
35480 }
35481 }

```

(End of definition for `\text_map_function:nN` and others. These functions are documented on page 307.)

94.4 Inline mappings

The standard non-expandable inline version.

```

\text_map_inline:nn
\text_words_map_inline:nn
35482 \cs_new_protected:Npn \text_map_inline:nn #1#2
35483 {
35484     \int_gincr:N \g__kernel_prg_map_int
35485     \cs_gset_protected:cpn
35486     { __text_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
35487     \exp_args:Nnc \text_map_function:nN {#1}
35488     { __text_map_ \int_use:N \g__kernel_prg_map_int :w }
35489     \prg_break_point:Nn \text_map_break:
35490     { \int_gdecr:N \g__kernel_prg_map_int }
35491 }
35492 \cs_new_protected:Npn \text_words_map_inline:nn #1#2
35493 {
35494     \int_gincr:N \g__kernel_prg_map_int
35495     \cs_gset_protected:cpn
35496     { __text_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
35497     \exp_args:Nnc \text_words_map_function:nN {#1}
35498     { __text_map_ \int_use:N \g__kernel_prg_map_int :w }
35499     \prg_break_point:Nn \text_map_break:
35500     { \int_gdecr:N \g__kernel_prg_map_int }
35501 }

```

(End of definition for `\text_map_inline:nn` and `\text_words_map_inline:nn`. These functions are documented on page 307.)

35502 `\</code>`

Chapter 95

l3text-purify implementation

```
35503 <*code>
35504 <@@=text>
```

95.1 Purifying text

```
\__text_if_recursion_tail_stop:N
```

Functions to query recursion quarks.

```
35505 \__kernel_quark_new_test:N \__text_if_recursion_tail_stop:N
```

(End of definition for __text_if_recursion_tail_stop:N.)

```
\text_purify:n
```

As in the other parts of the module, we start off with a standard “action” loop, with expansion applied up-front.

```
\text_purify:V
```

```
\text_purify:v
```

```
35506 \cs_new:Npn \text_purify:n #1
```

```
\__text_purify:n
```

```
{
```

```
\__text_purify_store:n
```

```
35507 \__kernel_exp_not:w \exp_after:wN
```

```
\__text_purify_store:nw
```

```
{
```

```
\__text_purify_end:w
```

```
\exp:w
```

```
\__text_purify_loop:w
```

```
35510 \exp_args:Ne \__text_purify:n
```

```
\__text_purify_group:n
```

```
35511 { \text_expand:n {#1} }
```

```
\__text_purify_space:w
```

```
}
```

```
\__text_purify_N_type:N
```

```
35512 }
```

```
\__text_purify_N_type_aux:N
```

```
35513 }
```

```
\_text_purify_math_search:NNN
```

```
35514 }
```

```
\_text_purify_math_start:NNw
```

```
35515 \cs_generate_variant:Nn \text_purify:n { V , v }
```

```
\__text_purify_math_store:n
```

```
35516 \cs_new:Npn \__text_purify:n #1
```

```
\__text_purify_math_store:nw
```

```
{
```

```
\__text_purify_math_end:w
```

```
35517 \group_align_safe_begin:
```

```
\__text_purify_math_loop:NNw
```

```
}
```

As for expansion, collect up the tokens for future use.

```
\_text_purify_math_N_type:NNN
```

```
35518 \__text_purify_loop:w #1
```

```
\_text_purify_math_group:NNn
```

```
\q__text_recursion_stop
```

```
\_text_purify_math_space:NNw
```

```
35519 \__text_purify_result:n { }
```

```
\__text_purify_math_cmd:N
```

```
35520 }
```

```
\__text_purify_math_cmd:NN
```

```
35521 \cs_new:Npn \__text_purify_store:nw #1#2 \__text_purify_result:n #3
```

```
\__text_purify_math_cmd:Nn
```

```
35522 { #2 \__text_purify_result:n { #3 #1 } }
```

```
\__text_purify_replace:N
```

```
{
```

```
\_text_purify_replace_auxi:n
```

```
35523 \group_align_safe_end:
```

```
\_text_purify_replace_auxi:v
```

```
\_text_purify_replace_auxii:n
```

```
}
```

```
\__text_purify_expand:N
```

```
\__text_purify_protect:N
```

```
\__text_purify_encoding:N
```

```
\_text_purify_encoding_escape:NN
```

```

35530 \exp_end:
35531 #2
35532 }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

35533 \cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
35534 {
35535   \tl_if_head_is_N_type:nTF {#1}
35536   { \__text_purify_N_type:N }
35537   {
35538     \tl_if_head_is_group:nTF {#1}
35539     { \__text_purify_group:n }
35540     { \__text_purify_space:w }
35541   }
35542   #1 \q__text_recursion_stop
35543 }
35544 \cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
35545 \exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
35546 {
35547   \__text_purify_store:n { ~ }
35548   \__text_purify_loop:w
35549 }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

35550 \cs_new:Npn \__text_purify_N_type:N #1
35551 {
35552   \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
35553   \__text_purify_N_type_aux:N #1
35554 }
35555 \cs_new:Npn \__text_purify_N_type_aux:N #1
35556 {
35557   \exp_after:wN \__text_purify_math_search:NNN
35558   \exp_after:wN #1 \l_text_math_delims_tl
35559   \q__text_recursion_tail ?
35560   \q__text_recursion_stop
35561 }
35562 \cs_new:Npn \__text_purify_math_search:NNN #1#2#3
35563 {
35564   \__text_if_q_recursion_tail_stop_do:Nn #2
35565   { \__text_purify_math_cmd:N #1 }
35566   \token_if_eq_meaning:NNTF #1 #2
35567   {
35568     \__text_use_i_delimit_by_q_recursion_stop:nw
35569     { \__text_purify_math_start:NNw #2 #3 }
35570   }
35571   { \__text_purify_math_search:NNN #1 }
35572 }
35573 \cs_new:Npn \__text_purify_math_start:NNw #1#2#3 \q__text_recursion_stop
35574 {
35575   \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
35576   \__text_purify_math_result:n { }
35577 }

```

```

35578 \cs_new:Npn \__text_purify_math_store:n #1
35579 { \__text_purify_math_store:nw {#1} }
35580 \cs_new:Npn \__text_purify_math_store:nw #1#2 \__text_purify_math_result:n #3
35581 { #2 \__text_purify_math_result:n { #3 #1 } }
35582 \cs_new:Npn \__text_purify_math_end:w #1 \__text_purify_math_result:n #2
35583 {
35584   \__text_purify_store:n { $ #2 $ }
35585   \__text_purify_loop:w #1
35586 }
35587 \cs_new:Npn \__text_purify_math_stop:Nw #1 \__text_purify_math_result:n #2
35588 {
35589   \__text_purify_store:n {#1#2}
35590   \__text_purify_end:w
35591 }
35592 \cs_new:Npn \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
35593 {
35594   \tl_if_head_is_N_type:nTF {#3}
35595   { \__text_purify_math_N_type:NNN }
35596   {
35597     \tl_if_head_is_group:nTF {#3}
35598     { \__text_purify_math_group:NNn }
35599     { \__text_purify_math_space:NNw }
35600   }
35601   #1#2#3 \q__text_recursion_stop
35602 }
35603 \cs_new:Npn \__text_purify_math_N_type:NNN #1#2#3
35604 {
35605   \__text_if_q_recursion_tail_stop_do:Nn #3
35606   { \__text_purify_math_stop:Nw #1 }
35607   \token_if_eq_meaning:NNTF #3 #2
35608   { \__text_purify_math_end:w }
35609   {
35610     \__text_purify_math_store:n {#3}
35611     \__text_purify_math_loop:NNw #1#2
35612   }
35613 }
35614 \cs_new:Npn \__text_purify_math_group:NNn #1#2#3
35615 {
35616   \__text_purify_math_store:n { {#3} }
35617   \__text_purify_math_loop:NNw #1#2
35618 }
35619 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_purify_math_space:NNw
35620 \exp_after:wN # \exp_after:wN 1
35621 \exp_after:wN # \exp_after:wN 2 \c_space_tl
35622 {
35623   \__text_purify_math_store:n { ~ }
35624   \__text_purify_math_loop:NNw #1#2
35625 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

35626 \cs_new:Npn \__text_purify_math_cmd:N #1
35627 {
35628   \exp_after:wN \__text_purify_math_cmd:NN \exp_after:wN #1
35629   \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
35630 }

```

```

35631 \cs_new:Npn \__text_purify_math_cmd:NN #1#2
35632 {
35633   \__text_if_q_recursion_tail_stop_do:Nn #2
35634   { \__text_purify_replace:N #1 }
35635   \cs_if_eq:NNTF #2 #1
35636   {
35637     \__text_use_i_delimit_by_q_recursion_stop:nw
35638     { \__text_purify_math_cmd:n }
35639   }
35640   { \__text_purify_math_cmd:NN #1 }
35641 }
35642 \cs_new:Npn \__text_purify_math_cmd:n #1
35643 { \__text_purify_math_end:w \__text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L^AT_EX 2_ε \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance. Notice that we filter out implicit begin/end group tokens to avoid issues.

```

35644 \cs_new:Npn \__text_purify_replace:N #1
35645 {
35646   \bool_lazy_and:nnTF
35647   { \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _tl } }
35648   {
35649     \bool_lazy_or_p:nn
35650     { \token_if_cs_p:N #1 }
35651     { \token_if_active_p:N #1 }
35652   }
35653   {
35654     \__text_purify_replace_auxi:v
35655     { l__text_purify_ \token_to_str:N #1 _tl }
35656   }
35657   {
35658     \bool_lazy_or:nnTF
35659     { \token_if_group_begin_p:N #1 }
35660     { \token_if_group_end_p:N #1 }
35661     { \__text_purify_loop:w }
35662     {
35663       \exp_args:Ne \__text_purify_replace_auxii:n
35664       { \__text_token_to_explicit:N #1 }
35665     }
35666   }
35667 }
35668 \cs_new:Npn \__text_purify_replace_auxi:n #1 { \__text_purify_loop:w #1 }
35669 \cs_generate_variant:Nn \__text_purify_replace_auxi:n { v }
35670 \cs_new:Npn \__text_purify_replace_auxii:n #1
35671 {
35672   \token_if_cs:NNTF #1
35673   { \__text_purify_expand:N #1 }
35674   {
35675     \__text_purify_store:n {#1}
35676     \__text_purify_loop:w
35677   }

```

```

35678 }
35679 \cs_new:Npn \__text_purify_expand:N #1
35680 {
35681   \str_if_eq:nnTF {#1} { \protect }
35682     { \__text_purify_protect:N }
35683     { \__text_purify_encoding:N #1 }
35684 }

```

We grab the token immediately after a `\protect` with the expectation that it is N-type. Whilst it shouldn't happen, this could be end-of-loop so there's a safety test for that. If there is some purification equivalent, we use it, but otherwise simply drop the token.

```

35685 \cs_new:Npn \__text_purify_protect:N #1
35686 {
35687   \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
35688   \cs_if_exist:cTF { l__text_purify_ \token_to_str:N #1 _tl }
35689     {
35690       \__text_purify_replace_auxi:v
35691       { l__text_purify_ \token_to_str:N #1 _tl }
35692     }
35693     { \__text_purify_loop:w }
35694 }

```

Handle encoding commands, as detailed for expansion.

```

35695 \cs_new:Npn \__text_purify_encoding:N #1
35696 {
35697   \bool_lazy_or:nnTF
35698     { \cs_if_eq_p:NN #1 \@current@cmd }
35699     { \cs_if_eq_p:NN #1 \@changed@cmd }
35700     { \__text_purify_encoding_escape:NN }
35701     {
35702       \__text_if_expandable:NTF #1
35703       { \exp_after:wN \__text_purify_loop:w #1 }
35704       { \__text_purify_loop:w }
35705     }
35706 }
35707 \cs_new:Npn \__text_purify_encoding_escape:NN #1#2
35708 {
35709   \__text_purify_store:n {#1}
35710   \__text_purify_loop:w
35711 }

```

(End of definition for `\text_purify:n` and others. This function is documented on page 306.)

```

\text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Ne

```

```

35712 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
35713 {
35714   \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
35715   \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
35716 }
35717 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Ne }

```

(End of definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 306.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

35718 \tl_map_inline:nn

```

```

35719 {
35720   \fontencoding
35721   \fontfamily
35722   \fontseries
35723   \fontshape
35724 }
35725 { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
35726 \text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
35727 \text_declare_purify_equivalent:Nn \selectfont { }
35728 \text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
35729 \exp_args:Nc \text_declare_purify_equivalent:Nn
35730 { @protected@testopt } { \use_none:nnn }

```

Environments have to be handled by pure expansion.

`\__text_end_env:n`

```

35731 \text_declare_purify_equivalent:Nn \begin { \use:c }
35732 \text_declare_purify_equivalent:Nn \end { \__text_end_env:n }
35733 \cs_new:Npn \__text_end_env:n #1 { \cs:w end #1 \cs_end: }

```

(End of definition for __text_end_env:n.)

Some common symbols and similar ideas.

```

35734 \text_declare_purify_equivalent:Nn \ { }
35735 \tl_map_inline:nn
35736 { \{ \} \# \$ \% \_ }
35737 { \text_declare_purify_equivalent:Ne #1 { \cs_to_str:N #1 } }

```

Cross-referencing.

```

35738 \text_declare_purify_equivalent:Nn \label { \use_none:n }

```

Spaces.

```

35739 \group_begin:
35740 \char_set_catcode_active:N \~
35741 \use:n
35742 {
35743   \group_end:
35744   \text_declare_purify_equivalent:Ne ~ { \c_space_tl }
35745 }
35746 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
35747 \text_declare_purify_equivalent:Nn \ { ~ }
35748 \text_declare_purify_equivalent:Nn \, { ~ }

```

95.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is `\SS`, which gets converted to two letters. (At some stage an alternative version can presumably be added to `babel` or similar.)

```

35749 \cs_set_protected:Npn \__text_loop:Nn #1#2
35750 {
35751   \quark_if_recursion_tail_stop:N #1
35752   \text_declare_purify_equivalent:Ne #1

```

```

35753     {
35754         \codepoint_generate:nn {"#2}
35755         { \char_value_catcode:n {"#2} }
35756     }
35757     \__text_loop:Nn
35758 }
35759 \__text_loop:Nn
35760 \AA { 00C5 }
35761 \AE { 00C6 }
35762 \DH { 00D0 }
35763 \DJ { 0110 }
35764 \IJ { 0132 }
35765 \L { 0141 }
35766 \NG { 014A }
35767 \O { 00D8 }
35768 \OE { 0152 }
35769 \TH { 00DE }
35770 \aa { 00E5 }
35771 \ae { 00E6 }
35772 \dh { 00F0 }
35773 \dj { 0111 }
35774 \i { 0131 }
35775 \j { 0237 }
35776 \ij { 0132 }
35777 \l { 0142 }
35778 \ng { 014B }
35779 \o { 00F8 }
35780 \oe { 0153 }
35781 \ss { 00DF }
35782 \th { 00FE }
35783 \q_recursion_tail ?
35784 \q_recursion_stop
35785 \text_declare_purify_equivalent:Nn \SS { SS }

```

__text_purify_accent:NN Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

35786 \cs_new:Npn \__text_purify_accent:NN #1#2
35787 {
35788     \cs_if_exist:cTF
35789     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _t1 }
35790     {
35791         \exp_not:v
35792         { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _t1 }
35793     }
35794     {
35795         \exp_not:n {"#2}
35796         \exp_not:v { c__text_purify_ \token_to_str:N #1 _t1 }
35797     }
35798 }
35799 \tl_map_inline:nn { \' \' ^ ~ \= \u \. \" \r \H \v \d \c \k \b \t }
35800 { \text_declare_purify_equivalent:Nn #1 { \__text_purify_accent:NN #1 } }

```


First set up the combining accents.

```

35801 \group_begin:
35802   \cs_set_protected:Npn \__text_loop:Nn #1#2
35803   {
35804     \quark_if_recursion_tail_stop:N #1
35805     \tl_const:ce { c__text_purify_ \token_to_str:N #1 _tl }
35806     { \codepoint_generate:nn {"#2} { \char_value_catcode:n { "#2 } } }
35807     \__text_loop:Nn
35808   }
35809   \__text_loop:Nn
35810   \' { 0300 }
35811   \' { 0301 }
35812   \^ { 0302 }
35813   \~ { 0303 }
35814   \= { 0304 }
35815   \u { 0306 }
35816   \. { 0307 }
35817   \" { 0308 }
35818   \r { 030A }
35819   \H { 030B }
35820   \v { 030C }
35821   \d { 0323 }
35822   \c { 0327 }
35823   \k { 0328 }
35824   \b { 0331 }
35825   \t { 0361 }
35826   \q_recursion_tail { }
35827   \q_recursion_stop

```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

35828   \cs_set_protected:Npn \__text_loop:NNn #1#2#3
35829   {
35830     \quark_if_recursion_tail_stop:N #1
35831     \tl_const:ce
35832     { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
35833     { \codepoint_generate:nn {"#3} { \char_value_catcode:n { "#3 } } }
35834     \__text_loop:NNn
35835   }
35836   \__text_loop:NNn
35837   \' A { 00C0 }
35838   \' A { 00C1 }
35839   \^ A { 00C2 }
35840   \~ A { 00C3 }
35841   \" A { 00C4 }
35842   \r A { 00C5 }
35843   \c C { 00C7 }
35844   \' E { 00C8 }
35845   \' E { 00C9 }
35846   \^ E { 00CA }
35847   \" E { 00CB }
35848   \' I { 00CC }

```

35849	\’ I	{ 00CD }
35850	\^ I	{ 00CE }
35851	\" I	{ 00CF }
35852	\~ N	{ 00D1 }
35853	\‘ O	{ 00D2 }
35854	\’ O	{ 00D3 }
35855	\^ O	{ 00D4 }
35856	\~ O	{ 00D5 }
35857	\" O	{ 00D6 }
35858	\‘ U	{ 00D9 }
35859	\’ U	{ 00DA }
35860	\^ U	{ 00DB }
35861	\" U	{ 00DC }
35862	\’ Y	{ 00DD }
35863	\‘ a	{ 00E0 }
35864	\’ a	{ 00E1 }
35865	\^ a	{ 00E2 }
35866	\~ a	{ 00E3 }
35867	\" a	{ 00E4 }
35868	\r a	{ 00E5 }
35869	\c c	{ 00E7 }
35870	\‘ e	{ 00E8 }
35871	\’ e	{ 00E9 }
35872	\^ e	{ 00EA }
35873	\" e	{ 00EB }
35874	\‘ i	{ 00EC }
35875	\‘ \i	{ 00EC }
35876	\’ i	{ 00ED }
35877	\’ \i	{ 00ED }
35878	\^ i	{ 00EE }
35879	\^ \i	{ 00EE }
35880	\" i	{ 00EF }
35881	\" \i	{ 00EF }
35882	\~ n	{ 00F1 }
35883	\‘ o	{ 00F2 }
35884	\’ o	{ 00F3 }
35885	\^ o	{ 00F4 }
35886	\~ o	{ 00F5 }
35887	\" o	{ 00F6 }
35888	\‘ u	{ 00F9 }
35889	\’ u	{ 00FA }
35890	\^ u	{ 00FB }
35891	\" u	{ 00FC }
35892	\’ y	{ 00FD }
35893	\" y	{ 00FF }
35894	\= A	{ 0100 }
35895	\= a	{ 0101 }
35896	\u A	{ 0102 }
35897	\u a	{ 0103 }
35898	\k A	{ 0104 }
35899	\k a	{ 0105 }
35900	\’ C	{ 0106 }
35901	\’ c	{ 0107 }
35902	\^ C	{ 0108 }

35903	\^ c	{ 0109 }
35904	\. C	{ 010A }
35905	\. c	{ 010B }
35906	\v C	{ 010C }
35907	\v c	{ 010D }
35908	\v D	{ 010E }
35909	\v d	{ 010F }
35910	\= E	{ 0112 }
35911	\= e	{ 0113 }
35912	\u E	{ 0114 }
35913	\u e	{ 0115 }
35914	\. E	{ 0116 }
35915	\. e	{ 0117 }
35916	\k E	{ 0118 }
35917	\k e	{ 0119 }
35918	\v E	{ 011A }
35919	\v e	{ 011B }
35920	\^ G	{ 011C }
35921	\^ g	{ 011D }
35922	\u G	{ 011E }
35923	\u g	{ 011F }
35924	\. G	{ 0120 }
35925	\. g	{ 0121 }
35926	\c G	{ 0122 }
35927	\c g	{ 0123 }
35928	\^ H	{ 0124 }
35929	\^ h	{ 0125 }
35930	\~ I	{ 0128 }
35931	\~ i	{ 0129 }
35932	\~ \i	{ 0129 }
35933	\= I	{ 012A }
35934	\= i	{ 012B }
35935	\= \i	{ 012B }
35936	\u I	{ 012C }
35937	\u i	{ 012D }
35938	\u \i	{ 012D }
35939	\k I	{ 012E }
35940	\k i	{ 012F }
35941	\k \i	{ 012F }
35942	\. I	{ 0130 }
35943	\^ J	{ 0134 }
35944	\^ j	{ 0135 }
35945	\^ \j	{ 0135 }
35946	\c K	{ 0136 }
35947	\c k	{ 0137 }
35948	\' L	{ 0139 }
35949	\' l	{ 013A }
35950	\c L	{ 013B }
35951	\c l	{ 013C }
35952	\v L	{ 013D }
35953	\v l	{ 013E }
35954	\. L	{ 013F }
35955	\. l	{ 0140 }
35956	\' N	{ 0143 }

35957	\' n	{ 0144 }
35958	\c N	{ 0145 }
35959	\c n	{ 0146 }
35960	\v N	{ 0147 }
35961	\v n	{ 0148 }
35962	\= O	{ 014C }
35963	\= o	{ 014D }
35964	\u O	{ 014E }
35965	\u o	{ 014F }
35966	\H O	{ 0150 }
35967	\H o	{ 0151 }
35968	\' R	{ 0154 }
35969	\' r	{ 0155 }
35970	\c R	{ 0156 }
35971	\c r	{ 0157 }
35972	\v R	{ 0158 }
35973	\v r	{ 0159 }
35974	\' S	{ 015A }
35975	\' s	{ 015B }
35976	\^ S	{ 015C }
35977	\^ s	{ 015D }
35978	\c S	{ 015E }
35979	\c s	{ 015F }
35980	\v S	{ 0160 }
35981	\v s	{ 0161 }
35982	\c T	{ 0162 }
35983	\c t	{ 0163 }
35984	\v T	{ 0164 }
35985	\v t	{ 0165 }
35986	\~ U	{ 0168 }
35987	\~ u	{ 0169 }
35988	\= U	{ 016A }
35989	\= u	{ 016B }
35990	\u U	{ 016C }
35991	\u u	{ 016D }
35992	\r U	{ 016E }
35993	\r u	{ 016F }
35994	\H U	{ 0170 }
35995	\H u	{ 0171 }
35996	\k U	{ 0172 }
35997	\k u	{ 0173 }
35998	\^ W	{ 0174 }
35999	\^ w	{ 0175 }
36000	\^ Y	{ 0176 }
36001	\^ y	{ 0177 }
36002	\" Y	{ 0178 }
36003	\' Z	{ 0179 }
36004	\' z	{ 017A }
36005	\. Z	{ 017B }
36006	\. z	{ 017C }
36007	\v Z	{ 017D }
36008	\v z	{ 017E }
36009	\v A	{ 01CD }
36010	\v a	{ 01CE }

```

36011 \v I { 01CF }
36012 \v \i { 01D0 }
36013 \v i { 01D0 }
36014 \v O { 01D1 }
36015 \v o { 01D2 }
36016 \v U { 01D3 }
36017 \v u { 01D4 }
36018 \v G { 01E6 }
36019 \v g { 01E7 }
36020 \v K { 01E8 }
36021 \v k { 01E9 }
36022 \k O { 01EA }
36023 \k o { 01EB }
36024 \v \j { 01F0 }
36025 \v j { 01F0 }
36026 \' G { 01F4 }
36027 \' g { 01F5 }
36028 \' N { 01F8 }
36029 \' n { 01F9 }
36030 \' \AE { 01FC }
36031 \' \ae { 01FD }
36032 \' \O { 01FE }
36033 \' \o { 01FF }
36034 \v H { 021E }
36035 \v h { 021F }
36036 \. A { 0226 }
36037 \. a { 0227 }
36038 \c E { 0228 }
36039 \c e { 0229 }
36040 \. O { 022E }
36041 \. o { 022F }
36042 \= Y { 0232 }
36043 \= y { 0233 }
36044 \q_recursion_tail ? { }
36045 \q_recursion_stop
36046 \group_end:

(End of definition for \_text_purify_accent:NN.)

36047 \</code>

```

Chapter 96

l3text-utils implementation

```
36048 (*code)
```

96.1 Parsing BCP 47 strings

```
36049 (@@=text_bcp)
```

For a reference implementation in JavaScript, see <https://github.com/woorm/bcp-47/>. This gives clear details of the overall algorithm needed.

\c__text_bcp_normal_prop There are a small number of non-standard tags which are grandfathered into the current standard. Here, we set up a mapping to the equivalent standard version, which allows us to avoid complexity. The commented lines have no equivalent I can track down at the moment! The prop is made into the linked form as this gains efficiency in the lookup.

```
36050 \prop_const_from_keyval:Nn \c__text_bcp_normal_prop
36051 {
36052   en-gb-oed    = en-gb-oxendict ,
36053   i-ami        = ami ,
36054   i-bnn        = bnn ,
36055   % i-default  = ,
36056   % i-enochian = ,
36057   i-hak        = hak ,
36058   i-klinton    = tlh ,
36059   i-lux        = lb ,
36060   % i-mingo    = ,
36061   i-navajo     = nv ,
36062   i-pwn        = pwn ,
36063   i-tao        = tao ,
36064   i-tay        = tay ,
36065   i-tsu        = tsu ,
36066   sgn-be-fr    = sfb ,
36067   sgn-be-nl    = vgt ,
36068   sgn-ch-de    = sgg ,
36069   art-lojban   = jbo ,
36070   % cel-gaulish = ,
36071   no-bok       = nb ,
36072   no-nyn       = nn ,
36073   zh-guoyu     = cmn ,
36074   zh-hakka     = hak ,
36075   % zh-min     = ,
```

```

36076     zh-min-nan = nan ,
36077     zh-xiang   = hsn
36078   }
36079 \prop_make_linked:N \c__text_bcp_normal_prop

```

(End of definition for \c__text_bcp_normal_prop.)

\text_bcp_parse:n

Before we get to the business end of the parse, we need to deal with the special cases: entirely blank input or one of the non-standard inputs above. We also want to deal with a string not a token list, but do that once any replacement is sorted.

```

36080 \cs_new:Npn \text_bcp_parse:n #1
36081 {
36082   \tl_if_blank:nTF {#1}
36083     { \msg_expandable_error:nn { text } { bcp-blank } }
36084     { \exp_args:Ne \__text_bcp_parse_auxi:n { \str_casefold:n {#1} } }
36085   }
36086 \cs_new:Npn \__text_bcp_parse_auxi:n #1
36087 {
36088   \exp_args:Ne \__text_bcp_parse_auxii:nn
36089     { \prop_item:Nn \c__text_bcp_normal_prop {#1} }
36090     {#1}
36091   }
36092 \cs_new:Npn \__text_bcp_parse_auxii:nn #1#2
36093 {
36094   \tl_if_blank:nTF {#1}
36095     { \__text_bcp_parse_auxiii:n {#2} }
36096     { \__text_bcp_parse_auxiii:n {#1} }
36097   }
36098 \cs_new:Npn \__text_bcp_parse_auxiii:n #1
36099 {
36100   \__text_bcp_parse_auxiv:w #1 - \q_recursion_tail - \q_recursion_stop
36101   }
36102 \cs_new:Npn \__text_bcp_parse_auxiv:w #1 -
36103 {
36104   \int_compare:nTF { 1 < \__text_bcp_parse_count:n {#1} < 4 }
36105     {
36106       {#1}
36107       \__text_bcp_parse_auxv:w
36108     }
36109     { \msg_expandable_error:nn { text } { bcp-invalid-lang } }
36110   }

```

The main loop is set up to allow us to iterate over each block, separated by a - token. The first block *must* be the language, which can be either two or three characters: this is an easy test. There doesn't have to be any other input, so things could well stop here. If not, we need to know how many characters are in the next block to proceed.

We will see versions of this several times. We know that there are a number of valid subtag types at this point, differentiated by their length. (A length of zero is never valid, but we do not special case it in the counting code as it's quite unlikely.) There's therefore a split to choose the appropriate subtag parser.

```

36111 \cs_new:Npn \__text_bcp_parse_auxv:w #1 -
36112 {
36113   \quark_if_recursion_tail_stop_do:nn {#1}

```

```

36114     { { } { } { } { } { } { } }
36115 \int_case:nnF { \__text_bcp_parse_count:n {#1} }
36116     {
36117         { 1 } { { } { } { } { } } \__text_bcp_parse_ext:n }
36118         { 2 } { { } { } { } \__text_bcp_parse_region:n }
36119         { 3 } { \__text_bcp_parse_extlang:n }
36120         { 4 } { { } \__text_bcp_parse_auxvi:n }
36121         { 5 } { { } { } { } \__text_bcp_parse_variant:n }
36122         { 6 } { { } { } { } \__text_bcp_parse_variant:n }
36123         { 7 } { { } { } { } \__text_bcp_parse_variant:n }
36124         { 8 } { { } { } { } \__text_bcp_parse_variant:n }
36125     }
36126     {
36127         \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36128         \use_none_delimit_by_q_recursion_stop:w
36129     }
36130     {#1}
36131 }

```

There is one case that cannot be determined purely on block length. Both variants and scripts can be made up of four characters, but for variants the first character has to be a digit.

```

36132 \cs_new:Npn \__text_bcp_parse_auxvi:n #1
36133 { \__text_bcp_parse_auxvii:NNNN #1 }
36134 \cs_new:Npn \__text_bcp_parse_auxvii:NNNN #1#2#3#4
36135 {
36136     \bool_lazy_or:nnTF
36137     { \int_compare_p:nNn {'#1} < { '0 } }
36138     { \int_compare_p:nNn {'#1} > { '9 } }
36139     { \__text_bcp_parse_script:n }
36140     { { } { } \__text_bcp_parse_variant:n }
36141     {#1#2#3#4}
36142 }

```

The first block allowed after the language is “extended language”, which can have up to three entries of three characters.

```

36143 \cs_new:Npn \__text_bcp_parse_extlang:n #1
36144 { \__text_bcp_parse_extlang:nw { {#1} } }
36145 \cs_new:Npn \__text_bcp_parse_extlang:nw #1#2 -
36146 {
36147     \quark_if_recursion_tail_stop_do:nn {#2}
36148     { {#1} { } { } { } { } { } }
36149     \int_case:nnF { \__text_bcp_parse_count:n {#2} }
36150     {
36151         { 1 } { { {#1} } { } { } { } } \__text_bcp_parse_ext:n }
36152         { 2 } { { {#1} } { } \__text_bcp_parse_region:n }
36153         { 3 } { \__text_bcp_parse_extlang:nn {#1} }
36154         { 4 } { { {#1} } \__text_bcp_parse_auxvi:n }
36155         { 5 } { { {#1} } { } { } \__text_bcp_parse_variant:n }
36156         { 6 } { { {#1} } { } { } \__text_bcp_parse_variant:n }
36157         { 7 } { { {#1} } { } { } \__text_bcp_parse_variant:n }
36158         { 8 } { { {#1} } { } { } \__text_bcp_parse_variant:n }
36159     }
36160     {
36161         \msg_expandable_error:nn { text } { bcp-invalid-subtag }

```



```

36162         \use_none_delimit_by_q_recursion_stop:w
36163     }
36164     {#2}
36165 }
36166 \cs_new:Npn \__text_bcp_parse_extlang:nn #1#2
36167 {
36168     \int_compare:nNnTF { \tl_count:n {#1} } = 3
36169     {
36170         \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36171         \use_none_delimit_by_q_recursion_stop:w
36172     }
36173     { \__text_bcp_parse_extlang:nw { #1 {#2} } }
36174 }

```

The next valid block is a script: a single entry so not a lot to do.

```

36175 \cs_new:Npn \__text_bcp_parse_script:n #1
36176 {
36177     {#1}
36178     \__text_bcp_parse_script:w
36179 }
36180 \cs_new:Npn \__text_bcp_parse_script:w #1 -
36181 {
36182     \quark_if_recursion_tail_stop_do:nn {#1}
36183     { { } { } { } { } }
36184     \int_case:nnF { \__text_bcp_parse_count:n {#1} }
36185     {
36186         { 1 } { { } { } } \__text_bcp_parse_ext:n }
36187         { 2 } { \__text_bcp_parse_region:n }
36188         { 4 } { { } } \__text_bcp_parse_variant_chk:n }
36189         { 5 } { { } } \__text_bcp_parse_variant:n }
36190         { 6 } { { } } \__text_bcp_parse_variant:n }
36191         { 7 } { { } } \__text_bcp_parse_variant:n }
36192         { 8 } { { } } \__text_bcp_parse_variant:n }
36193     }
36194     {
36195         \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36196         \use_none_delimit_by_q_recursion_stop:w
36197     }
36198     {#1}
36199 }

```

Much the same story for the region: a single block with simply fewer possible blocks after it.

```

36200 \cs_new:Npn \__text_bcp_parse_region:n #1
36201 {
36202     {#1}
36203     \__text_bcp_parse_region:w
36204 }
36205 \cs_new:Npn \__text_bcp_parse_region:w #1 -
36206 {
36207     \quark_if_recursion_tail_stop_do:nn {#1}
36208     { { } { } { } }
36209     \int_case:nnF { \__text_bcp_parse_count:n {#1} }
36210     {
36211         { 1 } { { } } \__text_bcp_parse_ext:n }

```

```

36212     { 4 } { \__text_bcp_parse_variant_chk:n }
36213     { 5 } { \__text_bcp_parse_variant:n }
36214     { 6 } { \__text_bcp_parse_variant:n }
36215     { 7 } { \__text_bcp_parse_variant:n }
36216     { 8 } { \__text_bcp_parse_variant:n }
36217   }
36218   {
36219     \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36220     \use_none_delimit_by_q_recursion_stop:w
36221   }
36222   {#1}
36223 }

```

The same idea about a four-character block as we've already seen: to be a valid variant, it has to start with a digit. Unlike the earlier version, at this stage a script is not allowed, so anything except a leading digit is an error.

```

36224 \cs_new:Npn \__text_bcp_parse_variant_chk:n #1
36225 { \__text_bcp_parse_variant_chk:NNNN #1 }
36226 \cs_new:Npn \__text_bcp_parse_variant_chk:NNNN #1#2#3#4
36227 {
36228   \bool_lazy_or:nnTF
36229   { \int_compare_p:nNn {'#1} < { '0 } }
36230   { \int_compare_p:nNn {'#1} > { '9 } }
36231   {
36232     \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36233     \use_none_delimit_by_q_recursion_stop:w
36234   }
36235   { \__text_bcp_parse_variant:n }
36236   {#1#2#3#4}
36237 }

```

Variants form an open-ended list so a loop is required to handle this. At each step, the length of the next block (if present) needs to be checked: if it's a valid variant, keep collecting, otherwise it's an extension or an error.

```

36238 \cs_new:Npn \__text_bcp_parse_variant:n #1
36239 { \__text_bcp_parse_variant:nw { {#1} } }
36240 \cs_new:Npn \__text_bcp_parse_variant:nw #1#2 -
36241 {
36242   \quark_if_recursion_tail_stop_do:nn {#2}
36243   { {#1} { } { } }
36244   \int_case:nnF { \__text_bcp_parse_count:n {#2} }
36245   {
36246     { 1 } { \__text_bcp_parse_variant_end:nn }
36247     { 4 } { \__text_bcp_parse_variant_chk:nn }
36248     { 5 } { \__text_bcp_parse_variant:nn }
36249     { 6 } { \__text_bcp_parse_variant:nn }
36250     { 7 } { \__text_bcp_parse_variant:nn }
36251     { 8 } { \__text_bcp_parse_variant:nn }
36252   }
36253   {
36254     \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36255     \use_none_delimit_by_q_recursion_stop:w
36256   }
36257   {#1} {#2}
36258 }

```

```

36259 \cs_new:Npn \__text_bcp_parse_variant:nn #1#2
36260 { \__text_bcp_parse_variant:nw { #2 {#1} } }
36261 \cs_new:Npn \__text_bcp_parse_variant_chk:nn #1#2
36262 { \__text_bcp_parse_variant_chk:NNNNn #1 {#2} }
36263 \cs_new:Npn \__text_bcp_parse_variant_chk:NNNNn #1#2#3#4
36264 {
36265   \bool_lazy_or:nnTF
36266     { \int_compare_p:nNn {'#1} < { '0 } }
36267     { \int_compare_p:nNn {'#1} > { '9 } }
36268   {
36269     \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36270     \use_none_delimit_by_q_recursion_stop:w
36271   }
36272   { \__text_bcp_parse_variant:nn }
36273   {#1#2#3#4} {#2}
36274 }
36275 \cs_new:Npn \__text_bcp_parse_variant_end:nn #1#2
36276 {
36277   {#2}
36278   \__text_bcp_parse_ext:n {#1}
36279 }

```

There are only three possible valid one-letter blocks: the extensions **t** and **u**, and the private use marker **x**. All of these then allow an open-ended set of subtags, the only restriction being these cannot be one-letter other than after **x**. So we need to collect up quite a bit of information whilst allowing for the fact that only one **u** or **t** should occur.

```

36280 \cs_new:Npn \__text_bcp_parse_ext:n #1
36281 {
36282   \str_if_eq:nnTF {#1} { x }
36283   {
36284     { }
36285     \__text_bcp_parse_private:nw { }
36286   }
36287   { \__text_bcp_parse_ext:nN { } #1 }
36288 }

```

Test for a valid letter, then start collecting up or switch to the private use area. Each extension can only be given once, and the comparison needs to be case-insensitive, so there is a little work to do.

```

36289 \cs_new:Npn \__text_bcp_parse_ext:nN #1#2
36290 {
36291   \bool_lazy_or:nnTF
36292     { \str_if_eq_p:nn {#2} { t } }
36293     { \str_if_eq_p:nn {#2} { u } }
36294   {
36295     \bool_lazy_or:nnTF
36296       { \tl_if_head_eq_charcode_p:nN {#1} #2 }
36297       { \int_compare_p:nNn { \tl_count:n {#1} } > 2 }
36298     {
36299       \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36300       \use_none_delimit_by_q_recursion_stop:w
36301     }
36302     { \__text_bcp_parse_ext:nNnw {#1} #2 { } }
36303   }

```

```

36304     {
36305         \str_if_eq:nnTF {#2} { x }
36306         {
36307             {#1}
36308             \__text_bcp_parse_private:nw { }
36309         }
36310         {
36311             \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36312             \use_none_delimit_by_q_recursion_stop:w
36313         }
36314     }
36315 }

```

The loop for extensions: largely just collection with a test in case we find another extension or private use marker.

```

36316 \cs_new:Npn \__text_bcp_parse_ext:nNnw #1#2#3#4 -
36317 {
36318     \quark_if_recursion_tail_stop_do:nn {#4}
36319     {
36320         \str_if_empty:nTF {#3}
36321         { \msg_expandable_error:nn { text } { bcp-invalid-subtag } }
36322         { { #1 #2 {#3} } { } }
36323     }
36324     \int_compare:nTF { 1 < \__text_bcp_parse_count:n {#4} < 9 }
36325     { \__text_bcp_parse_ext:nNnw {#1} #2 { #3 {#4} } }
36326     {
36327         \int_compare:nNnTF { \__text_bcp_parse_count:n {#4} } = 1
36328         {
36329             \str_if_empty:nTF {#3}
36330             {
36331                 \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36332                 \use_none_delimit_by_q_recursion_stop:w
36333             }
36334             {
36335                 \use:e
36336                 {
36337                     \exp_not:n { \__text_bcp_parse_ext:nN { #1 #2 {#3} } }
36338                     \str_casefold:n {#4}
36339                 }
36340             }
36341         }
36342         {
36343             \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36344             \use_none_delimit_by_q_recursion_stop:w
36345         }
36346     }
36347 }

```

Private use area: all bets are off! The only tests here is we need at least one subtag, and they all need to be shorter than nine characters.

```

36348 \cs_new:Npn \__text_bcp_parse_private:nw #1#2 -
36349 {
36350     \quark_if_recursion_tail_stop_do:nn {#2}
36351     {
36352         \str_if_empty:nTF {#1}

```

```

36353         { \msg_expandable_error:nn { text } { bcp-invalid-subtag } }
36354         { {#1} }
36355     }
36356     \int_compare:nTF { 0 < \__text_bcp_parse_count:n {#2} < 9 }
36357     { \__text_bcp_parse_private:nw { #1 {#2} } }
36358     {
36359         \msg_expandable_error:nn { text } { bcp-invalid-subtag }
36360         \use_none_delimit_by_q_recursion_stop:w
36361     }
36362 }

```

As BCP 47 is largely specified in terms of number of characters, there is a need to count up repeatedly. Whilst `\tl_count:n` is reasonably fast, the predictable nature of the input here means we can use a slightly more focussed approach. We know that input can never be more than 8 characters, so can test for that, then get the character number by a simple expansion. This saves around half the tracing lines for typical input lengths.

```

36363 \cs_new:Npn \__text_bcp_parse_count:n #1
36364 {
36365     \__text_bcp_parse_count_auxi:w #1
36366     \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil \q_nil
36367     \q_stop {#1}
36368 }
36369 \cs_new:Npn \__text_bcp_parse_count_auxi:w #1#2#3#4#5#6#7#8#9
36370 {
36371     \quark_if_nil:NTF #9
36372     { \__text_bcp_parse_count_auxii:w }
36373     { \msg_expandable_error:nn { text } { bcp-invalid-lang } }
36374 }
36375 \cs_new:Npn \__text_bcp_parse_count_auxii:w #1 \q_stop #2
36376 { \__text_bcp_parse_count_auxiii:w #2 876543210 \q_stop }
36377 \cs_new:Npn \__text_bcp_parse_count_auxiii:w #1#2#3#4#5#6#7#8#9
36378 { \__text_bcp_parse_count_auxiv:N #9 }
36379 \cs_new:Npn \__text_bcp_parse_count_auxiv:N #1#2 \q_stop {#1}

```

(End of definition for `\text_bcp_parse:n` and others. This function is documented on page 309.)

```

36380 \msg_new:nnn { text } { bcp-blank }
36381 { Empty~input~for~BCP~47~decoding. }
36382 \msg_new:nnn { text } { bcp-invalid-lang }
36383 { Invalid~language~in~BCP~input. }
36384 \msg_new:nnn { text } { bcp-invalid-subtag }
36385 { Invalid~subtag~in~BCP~input. }
36386 </code>

```

Chapter 97

l3box implementation

```
36387 <*code>
36388 <@@=box>
```

97.1 Support code

`\__box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
36389 \cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
36390 \cs_new:Npn \__box_dim_eval:n #1
36391 { \__box_dim_eval:w #1 \scan_stop: }
```

(End of definition for __box_dim_eval:w and __box_dim_eval:n.)

97.2 Creating and initializing boxes

The following test files are used for this code: m3box001.lvt.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
\box_new:c 36392 \cs_new_protected:Npn \box_new:N #1
36393 {
36394   \__kernel_chk_if_free_cs:N #1
36395   \cs:w newbox \cs_end: #1
36396 }
36397 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a `<box>` register.

```
36398 \cs_new_protected:Npn \box_clear:N #1
\box_clear:N 36399 { \box_set_eq:NN #1 \c_empty_box }
36400 \cs_new_protected:Npn \box_gclear:N #1
\box_gclear:N 36401 { \box_gset_eq:NN #1 \c_empty_box }
36402 \cs_generate_variant:Nn \box_clear:N { c }
\box_gclear:c 36403 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```

\box_clear_new:N 36404 \cs_new_protected:Npn \box_clear_new:N #1
                  36405 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_clear_new:c 36406 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 36407 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\box_gclear_new:c 36408 \cs_generate_variant:Nn \box_clear_new:N { c }
                  36409 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

Assigning the contents of a box to be another box.

```

\box_set_eq:NN 36410 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:cN 36411 { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:Nc 36412 \cs_new_protected:Npn \box_gset_eq:NN #1#2
\box_set_eq:cc 36413 { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\box_gset_eq:NN 36414 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:cN 36415 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:Nc
\box_gset_eq:cc
\box_set_eq_drop:NN 36416 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
\box_set_eq_drop:cN 36417 { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:Nc 36418 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
\box_set_eq_drop:cc 36419 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_gset_eq_drop:NN 36420 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:cN 36421 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc
\box_if_exist_p:N 36422 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
\box_if_exist_p:c 36423 { TF , T , F , p }
\box_if_exist:NTF 36424 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
\box_if_exist:cTF 36425 { TF , T , F , p }

```

Assigning the contents of a box to be another box, then drops the original box.

Copies of the cs functions defined in l3basics.

97.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

\box_ht:N 36426 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:c 36427 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_dp:N 36428 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:c 36429 \cs_generate_variant:Nn \box_ht:N { c }
\box_wd:N 36430 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:c 36431 \cs_generate_variant:Nn \box_wd:N { c }

```

The $\backslash box\_ht:N$ and $\backslash box\_dp:N$ primitives do not expand but rather are suitable for use after $\backslash the$ or inside dimension expressions. Here we obtain the same behavior by using $\backslash \_box\_dim\_eval:n$ (basically $\backslash dimexpr$) rather than $\backslash dim\_eval:n$ (basically $\backslash the \backslash dimexpr$).

```

36432 \cs_new_protected:Npn \box_ht_plus_dp:N #1
36433 { \_box_dim_eval:n { \box_ht:N #1 + \box_dp:N #1 } }
36434 \cs_generate_variant:Nn \box_ht_plus_dp:N { c }

```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression $\#2$ is surrounded by parentheses to catch early termination.

```

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn

```

```

36435 \cs_new_protected:Npn \box_set_dp:Nn #1#2
36436 {
36437   \tex_setbox:D #1 = \tex_copy:D #1
36438   \box_dp:N #1 \__box_dim_eval:n {#2}
36439 }
36440 \cs_generate_variant:Nn \box_set_dp:Nn { c }
36441 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
36442 { \box_dp:N #1 \__box_dim_eval:n {#2} }
36443 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
36444 \cs_new_protected:Npn \box_set_ht:Nn #1#2
36445 {
36446   \tex_setbox:D #1 = \tex_copy:D #1
36447   \box_ht:N #1 \__box_dim_eval:n {#2}
36448 }
36449 \cs_generate_variant:Nn \box_set_ht:Nn { c }
36450 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
36451 { \box_ht:N #1 \__box_dim_eval:n {#2} }
36452 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
36453 \cs_new_protected:Npn \box_set_wd:Nn #1#2
36454 {
36455   \tex_setbox:D #1 = \tex_copy:D #1
36456   \box_wd:N #1 \__box_dim_eval:n {#2}
36457 }
36458 \cs_generate_variant:Nn \box_set_wd:Nn { c }
36459 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
36460 { \box_wd:N #1 \__box_dim_eval:n {#2} }
36461 \cs_generate_variant:Nn \box_gset_wd:Nn { c }

```

97.4 Using boxes

Using a `<box>`. These are just TeX primitives with meaningful names.

```

36462 \cs_new_eq:NN \box_use_drop:N \tex_box:D
\box_use_drop:c 36463 \cs_new_eq:NN \box_use:N \tex_copy:D
36464 \cs_generate_variant:Nn \box_use_drop:N { c }
\box_use:N 36465 \cs_generate_variant:Nn \box_use:N { c }
\box_use:c

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn 36466 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_right:nn 36467 { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
\box_move_up:nn 36468 \cs_new_protected:Npn \box_move_right:nn #1#2
\box_move_down:nn 36469 { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
36470 \cs_new_protected:Npn \box_move_up:nn #1#2
36471 { \tex_raise:D \__box_dim_eval:n {#1} #2 }
36472 \cs_new_protected:Npn \box_move_down:nn #1#2
36473 { \tex_lower:D \__box_dim_eval:n {#1} #2 }

```

97.5 Box conditionals

The primitives for testing if a `<box>` is empty/void or which type of box it is.

```

36474 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_hbox:N
\if_vbox:N
\if_box_empty:N

```



```

36475 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
36476 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D

```

```

\box_if_horizontal_p:N 36477 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:c 36478 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:NTF 36479 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_horizontal:cTF 36480 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_vertical_p:N 36481 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
\box_if_vertical_p:c 36482 { c } { p , T , F , TF }
\box_if_vertical:NTF 36483 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
\box_if_vertical:cTF 36484 { c } { p , T , F , TF }

```

Testing if a $\langle\text{box}\rangle$ is empty/void.

```

\box_if_empty_p:N 36485 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:c 36486 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:NTF 36487 \prg_generate_conditional_variant:Nnn \box_if_empty:N
\box_if_empty:cTF 36488 { c } { p , T , F , TF }

```

(End of definition for $\backslash\text{box\_new:N}$ and others. These functions are documented on page 311.)

97.6 The last box inserted

```

\box_set_to_last:N Set a box to the previous box.
\box_set_to_last:c 36489 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:N 36490 { \tex_setbox:D #1 \tex_lastbox:D }
\box_gset_to_last:c 36491 \cs_new_protected:Npn \box_gset_to_last:N #1
36492 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
36493 \cs_generate_variant:Nn \box_set_to_last:N { c }
36494 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End of definition for $\backslash\text{box\_set\_to\_last:N}$ and $\backslash\text{box\_gset\_to\_last:N}$. These functions are documented on page 314.)

97.7 Constant boxes

$\backslash\text{c\_empty\_box}$ A box we never use.

```

36495 \box_new:N \c_empty_box

```

(End of definition for $\backslash\text{c\_empty\_box}$. This variable is documented on page 314.)

97.8 Scratch boxes

```

\l_tmpa_box Scratch boxes.
\l_tmpb_box 36496 \box_new:N \l_tmpa_box
\g_tmpa_box 36497 \box_new:N \l_tmpb_box
\g_tmpb_box 36498 \box_new:N \g_tmpa_box
36499 \box_new:N \g_tmpb_box

```

(End of definition for $\backslash\text{l\_tmpa\_box}$ and others. These variables are documented on page 314.)

97.9 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.
`\box_show:c`
`\box_show:Nnn`
`\box_show:cnn`

```

36500 \cs_new_protected:Npn \box_show:N #1
36501 { \box_show:Nnn #1 \c_max_int \c_max_int }
36502 \cs_generate_variant:Nn \box_show:N { c }
36503 \cs_new_protected:Npn \box_show:Nnn #1#2#3
36504 { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
36505 \cs_generate_variant:Nn \box_show:Nnn { c }

```

(End of definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 314.)

`\box_log:N` Getting TeX to write to the log without interruption the run is done by altering the
`\box_log:c` interaction mode. For that, the ε -TeX extensions are needed.
`\box_log:Nnn`
`\box_log:cnn`
`\__box_log:nNnn`

```

36506 \cs_new_protected:Npn \box_log:N #1
36507 { \box_log:Nnn #1 \c_max_int \c_max_int }
36508 \cs_generate_variant:Nn \box_log:N { c }
36509 \cs_new_protected:Npn \box_log:Nnn
36510 { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
36511 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
36512 {
36513   \int_gset:Nn \tex_interactionmode:D { 0 }
36514   \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
36515   \int_gset:Nn \tex_interactionmode:D {#1}
36516 }
36517 \cs_generate_variant:Nn \box_log:Nnn { c }

```

(End of definition for `\box_log:N`, `\box_log:Nnn`, and `\__box_log:nNnn`. These functions are documented on page 314.)

`\__box_show:NNnn` The internal auxiliary to actually do the output uses a group to deal with breadth and
`\__box_show:NNff` depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline` and `\errorcontextlines` is used to control what appears in the terminal.

```

36518 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
36519 {
36520   \box_if_exist:NTF #2
36521   {
36522     \group_begin:
36523     \int_set:Nn \tex_showboxbreadth:D {#3}
36524     \int_set:Nn \tex_showboxdepth:D {#4}
36525     \int_set:Nn \tex_tracingonline:D {#1}
36526     \int_set:Nn \tex_errorcontextlines:D { -1 }
36527     \tex_showbox:D \use:n {#2}
36528   \group_end:
36529   }
36530   {
36531     \msg_error:nne { kernel } { variable-not-defined }
36532     { \token_to_str:N #2 }
36533   }

```

```

36534 }
36535 \cs_generate_variant:Nn \__box_show:NNnn { NNff }

```

(End of definition for __box_show:NNnn.)

97.10 Horizontal mode boxes

\hbox:n (The test suite for this command, and others in this file, is *m3box002.lvt*.)
Put a horizontal box directly into the input stream.

```

36536 \cs_new_protected:Npn \hbox:n #1
36537 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

```

(End of definition for \hbox:n. This function is documented on page 315.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
36538 \cs_new_protected:Npn \hbox_set:Nn #1#2
36539 {
36540   \tex_setbox:D #1 \tex_hbox:D
36541   { \color_group_begin: #2 \color_group_end: }
36542 }
36543 \cs_new_protected:Npn \hbox_gset:Nn #1#2
36544 {
36545   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
36546   { \color_group_begin: #2 \color_group_end: }
36547 }
36548 \cs_generate_variant:Nn \hbox_set:Nn { c }
36549 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End of definition for \hbox_set:Nn and \hbox_gset:Nn. These functions are documented on page 315.)

\hbox_set_to_wd:Nnn Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_set_to_wd:cn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cn
36550 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
36551 {
36552   \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
36553   { \color_group_begin: #3 \color_group_end: }
36554 }
36555 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
36556 {
36557   \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
36558   { \color_group_begin: #3 \color_group_end: }
36559 }
36560 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
36561 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End of definition for \hbox_set_to_wd:Nnn and \hbox_gset_to_wd:Nnn. These functions are documented on page 315.)

\hbox_set:Nw Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:
36562 \cs_new_protected:Npn \hbox_set:Nw #1
36563 {
36564   \tex_setbox:D #1 \tex_hbox:D
36565   \c_group_begin_token
36566   \color_group_begin:

```

```

36567 }
36568 \cs_new_protected:Npn \hbox_gset:Nw #1
36569 {
36570   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
36571   \c_group_begin_token
36572   \color_group_begin:
36573 }
36574 \cs_generate_variant:Nn \hbox_set:Nw { c }
36575 \cs_generate_variant:Nn \hbox_gset:Nw { c }
36576 \cs_new_protected:Npn \hbox_set_end:
36577 {
36578   \color_group_end:
36579   \c_group_end_token
36580 }
36581 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End of definition for `\hbox_set:Nw` and others. These functions are documented on page 315.)

`\hbox_set_to_wd:Nnw`
`\hbox_set_to_wd:cnw`
`\hbox_gset_to_wd:Nnw`
`\hbox_gset_to_wd:cnw`

Combining the above ideas.

```

36582 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
36583 {
36584   \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
36585   \c_group_begin_token
36586   \color_group_begin:
36587 }
36588 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
36589 {
36590   \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
36591   \c_group_begin_token
36592   \color_group_begin:
36593 }
36594 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
36595 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }

```

(End of definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 316.)

`\hbox_to_wd:nn`
`\hbox_to_zero:n`

Put a horizontal box directly into the input stream.

```

36596 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
36597 {
36598   \tex_hbox:D to \__box_dim_eval:n {#1}
36599   { \color_group_begin: #2 \color_group_end: }
36600 }
36601 \cs_new_protected:Npn \hbox_to_zero:n #1
36602 {
36603   \tex_hbox:D to \c_zero_dim
36604   { \color_group_begin: #1 \color_group_end: }
36605 }

```

(End of definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 315.)

`\hbox_overlap_center:n`
`\hbox_overlap_left:n`
`\hbox_overlap_right:n`

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

36606 \cs_new_protected:Npn \hbox_overlap_center:n #1

```

```

36607 { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
36608 \cs_new_protected:Npn \hbox_overlap_left:n #1
36609 { \hbox_to_zero:n { \tex_hss:D #1 } }
36610 \cs_new_protected:Npn \hbox_overlap_right:n #1
36611 { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End of definition for `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, and `\hbox_overlap_right:n`. These functions are documented on page 315.)

```

\hbox_unpack:N Unpacking a box and if requested also clear it.
\hbox_unpack:c 36612 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\hbox_unpack_drop:N 36613 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
\hbox_unpack_drop:c 36614 \cs_generate_variant:Nn \hbox_unpack:N { c }
36615 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }

```

(End of definition for `\hbox_unpack:N` and `\hbox_unpack_drop:N`. These functions are documented on page 316.)

97.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

```

\ vbox:n Put a vertical box directly into the input stream.
\ vbox_top:n 36616 \cs_new_protected:Npn \vbox:n #1
\ vbox:w 36617 { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
\ vbox_top:w 36618 \cs_new_protected:Npn \vbox_top:n #1
\ vbox_end: 36619 { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }
36620 \cs_new_protected:Npn \vbox:w
36621 { \tex_vbox:D \c_group_begin_token \color_group_begin: }
36622 \cs_new_protected:Npn \vbox_top:w
36623 { \tex_vtop:D \c_group_begin_token \color_group_begin: }
36624 \cs_new_protected:Npn \vbox_end:
36625 {
36626 \par
36627 \color_group_end:
36628 \c_group_end_token
36629 }

```

(End of definition for `\vbox:n` and others. These functions are documented on page 316.)

```

\l__box_delta_dim Needed for the vertical centering calc.
\l__box_voffset_dim 36630 \dim_new:N \l__box_delta_dim
36631 \dim_new:N \l__box_voffset_dim

```

(End of definition for `\l__box_delta_dim` and `\l__box_voffset_dim`.)

```

\ vbox_center:nw This is set up first as the :nw/:nn version as that's required to implement the :n one;
\ vbox_center:w contrast with the other two vertical placements, where we start with the simple case
\ vbox_center_to_ht:nw then build the more involved one. To avoid reading content multiple times, we use two
\__box_vcenter:nnw internals for the w and n versions, then use a common auxiliary at the calculation stage.
\__box_vcenter_aux: 36632 \cs_new_protected:Npn \vbox_center:nw
\ vbox_center:nn 36633 { \__box_vcenter:nnw { } }
\ vbox_center:n
\ vbox_center_to_ht:nn
\__box_vcenter:Nnnn
\__box_vcenter_calc:n

```

```

36634 \cs_new_protected:Npn \vbox_center:w
36635   { \__box_vcenter:nnw { } \c_zero_dim }
36636 \cs_new_protected:Npn \vbox_center_to_ht:nw #1
36637   { \__box_vcenter:nnw { to \__box_dim_eval:n {#1} } \c_zero_dim }

```

Starting the box with `\c_group_begin_token` avoids needing `\afterassignment` to get the end-of-group tokens in the correct place.

```

36638 \cs_new_protected:Npn \__box_vcenter:nnw #1#2
36639   {
36640     \dim_set:Nn \l__box_voffset_dim {#2}
36641     \tex_setbox:D \l__box_tmp_box \tex_vbox:D #1
36642     \c_group_begin_token
36643     \group_insert_after:N \__box_vcenter_aux:
36644     \color_group_begin:
36645   }
36646 \cs_new_protected:Npn \__box_vcenter_aux:
36647   { \__box_vcenter_calc:n \l__box_voffset_dim }
36648 \cs_new_protected:Npn \vbox_center:nn
36649   { \__box_vcenter:Nnnn \vbox_set:Nn { } }
36650 \cs_new_protected:Npn \vbox_center:n
36651   { \__box_vcenter:Nnnn \vbox_set:Nn { } \c_zero_dim }
36652 \cs_new_protected:Npn \vbox_center_to_ht:nn #1
36653   { \__box_vcenter:Nnnn \vbox_set_to_ht:Nnn { {#1} } \c_zero_dim }
36654 \cs_new_protected:Npn \__box_vcenter:Nnnn #1#2#3#4
36655   {
36656     #1 \l__box_tmp_box #2 {#4}
36657     \__box_vcenter_calc:n {#3}
36658   }

```

To get exactly the same outcomes as `\vcenter`, we need to implement *exactly* the approach used in `tex.web`:

```

delta:=height(v)+depth(v);
height(v):=axis_height(cur_size)+half(delta);
depth(v):=delta-height(v);

```

where

```

@p function half(@!x:integer):integer;
begin if odd(x) then half:=(x+1) div 2
else half:=x @!div 2;
end;

```

```

36659 \cs_new_protected:Npn \__box_vcenter_calc:n #1
36660   {
36661     \dim_set:Nn \l__box_delta_dim
36662       { \box_ht:N \l__box_tmp_box + \box_dp:N \l__box_tmp_box }
36663     \tex_ht:D \l__box_tmp_box \__box_dim_eval:n
36664     {
36665       \__box_dim_eval:n
36666       { \tex_ifodd:D \l__box_delta_dim 1sp + \tex_fi:D \l__box_delta_dim }
36667       / 2
36668       + (#1)
36669     }
36670     \tex_dp:D \l__box_tmp_box \__box_dim_eval:n
36671     { \l__box_delta_dim - \box_ht:N \l__box_tmp_box }

```

```

36672 \box_use_drop:N \l__box_tmp_box
36673 }

```

(End of definition for `\vbox_center:nw` and others. These functions are documented on page 317.)

```

\vbox_to_ht:nn Put a vertical box directly into the input stream.
\vbox_top_to_ht:nn
\__box_to_ht:Nnn
\vbox_to_ht:nw
\vbox_top_to_ht:nw
\__box_to_ht:Nnw
\vbox_to_zero:nn
36674 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
36675 { \__box_to_ht:Nnn \tex_vbox:D {#1} {#2} }
36676 \cs_new_protected:Npn \vbox_top_to_ht:nn #1#2
36677 { \__box_to_ht:Nnn \tex_vtop:D {#1} {#2} }
36678 \cs_new_protected:Npn \__box_to_ht:Nnn #1#2#3
36679 {
36680 #1 to \__box_dim_eval:n {#2}
36681 { \color_group_begin: #3 \par \color_group_end: }
36682 }
36683 \cs_new_protected:Npn \vbox_to_ht:nw #1
36684 { \__box_to_ht:Nnw \tex_vbox:D {#1} }
36685 \cs_new_protected:Npn \vbox_top_to_ht:nw #1
36686 { \__box_to_ht:Nnw \tex_vtop:D {#1} }
36687 \cs_new_protected:Npn \__box_to_ht:Nnw #1#2
36688 {
36689 #1 to \__box_dim_eval:n {#2}
36690 \c_group_begin_token
36691 \color_group_begin:
36692 }
36693 \cs_new_protected:Npn \vbox_to_zero:n #1
36694 { \vbox_to_ht:nn { Opt } {#1} }

```

(End of definition for `\vbox_to_ht:nn` and others. These functions are documented on page 317.)

```

\vbox:nw Mainly for consistency of language.
\vbox_top:nw
\__box_voffset:Nnnw
\__box_voffset_aux:
\vbox:nn
\vbox_top:nn
\__box_voffset:Nnnn
\__box_voffset_calc:n
36695 \cs_new_protected:Npn \vbox:nw
36696 { \__box_voffset:Nnnw \tex_vbox:D { } }
36697 \cs_new_protected:Npn \vbox_top:nw
36698 { \__box_voffset:Nnnw \tex_vtop:D { } }
36699 \cs_new_protected:Npn \__box_voffset:Nnnw #1#2#3
36700 {
36701 \dim_set:Nn \l__box_voffset_dim {#3}
36702 \tex_setbox:D \l__box_tmp_box #1 #2
36703 \c_group_begin_token
36704 \group_insert_after:N \__box_voffset_aux:
36705 \color_group_begin:
36706 }
36707 \cs_new_protected:Npn \__box_voffset_aux:
36708 { \__box_voffset_calc:n \l__box_voffset_dim }
36709 \cs_new_protected:Npn \vbox:nn
36710 { \__box_voffset:Nnnn \vbox_set:Nn { } }
36711 \cs_new_protected:Npn \vbox_top:nn
36712 { \__box_voffset:Nnnn \vbox_set_top:Nn { } }
36713 \cs_new_protected:Npn \__box_voffset:Nnnn #1#2#3#4
36714 {
36715 #1 \l__box_tmp_box #2 {#4}
36716 \__box_voffset_calc:n {#3}
36717 }
36718 \cs_new_protected:Npn \__box_voffset_calc:n #1

```

```

36719 {
36720   \box_set_ht:Nn \l__box_tmp_box
36721   { \box_ht:N \l__box_tmp_box + (#1) }
36722   \box_set_dp:Nn \l__box_tmp_box
36723   { \box_dp:N \l__box_tmp_box - (#1) }
36724   \box_use_drop:N \l__box_tmp_box
36725 }

```

(End of definition for `\vbox:nw` and others. These functions are documented on page 317.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.
`\vbox_set:cn` 36726 `\cs_new_protected:Npn \vbox_set:Nn #1#2`
`\vbox_gset:Nn` 36727 {
`\vbox_gset:cn` 36728 `\tex_setbox:D #1 \tex_vbox:D`
36729 `{ \color_group_begin: #2 \par \color_group_end: }`
36730 }
36731 `\cs_new_protected:Npn \vbox_gset:Nn #1#2`
36732 {
36733 `\tex_global:D \tex_setbox:D #1 \tex_vbox:D`
36734 `{ \color_group_begin: #2 \par \color_group_end: }`
36735 }
36736 `\cs_generate_variant:Nn \vbox_set:Nn { c }`
36737 `\cs_generate_variant:Nn \vbox_gset:Nn { c }`

(End of definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 317.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline
`\vbox_set_top:cn` of the first object in the box.
`\vbox_gset_top:Nn` 36738 `\cs_new_protected:Npn \vbox_set_top:Nn #1#2`
`\vbox_gset_top:cn` 36739 {
36740 `\tex_setbox:D #1 \tex_vtop:D`
36741 `{ \color_group_begin: #2 \par \color_group_end: }`
36742 }
36743 `\cs_new_protected:Npn \vbox_gset_top:Nn #1#2`
36744 {
36745 `\tex_global:D \tex_setbox:D #1 \tex_vtop:D`
36746 `{ \color_group_begin: #2 \par \color_group_end: }`
36747 }
36748 `\cs_generate_variant:Nn \vbox_set_top:Nn { c }`
36749 `\cs_generate_variant:Nn \vbox_gset_top:Nn { c }`

(End of definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 318.)

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn` 36750 `\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3`
`\vbox_gset_to_ht:Nnn` 36751 {
`\vbox_gset_to_ht:cnn` 36752 `\tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}`
36753 `{ \color_group_begin: #3 \par \color_group_end: }`
36754 }
36755 `\cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3`
36756 {
36757 `\tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}`
36758 `{ \color_group_begin: #3 \par \color_group_end: }`
36759 }


```

36760 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
36761 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End of definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 318.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.

```

\vbox_set:cw 36762 \cs_new_protected:Npn \vbox_set:Nw #1
\vbox_gset:Nw 36763 {
\vbox_gset:cw 36764   \tex_setbox:D #1 \tex_vbox:D
\vbox_set_end: 36765   \c_group_begin_token
\vbox_gset_end: 36766   \color_group_begin:
36767 }
36768 \cs_new_protected:Npn \vbox_gset:Nw #1
36769 {
36770   \tex_global:D \tex_setbox:D #1 \tex_vbox:D
36771   \c_group_begin_token
36772   \color_group_begin:
36773 }
36774 \cs_generate_variant:Nn \vbox_set:Nw { c }
36775 \cs_generate_variant:Nn \vbox_gset:Nw { c }
36776 \cs_new_eq:NN \vbox_set_end: \vbox_end:
36777 \cs_new_eq:NN \vbox_gset_end: \vbox_end:

```

(End of definition for `\vbox_set:Nw` and others. These functions are documented on page 318.)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.

```

\vbox_set_to_ht:cnw 36778 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
\vbox_gset_to_ht:Nnw 36779 {
\vbox_gset_to_ht:cnw 36780   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
36781   \c_group_begin_token
36782   \color_group_begin:
36783 }
36784 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
36785 {
36786   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
36787   \c_group_begin_token
36788   \color_group_begin:
36789 }
36790 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
36791 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End of definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 318.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.

```

\vbox_unpack:c 36792 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\vbox_unpack_drop:N 36793 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
\vbox_unpack_drop:c 36794 \cs_generate_variant:Nn \vbox_unpack:N { c }
36795 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }

```

(End of definition for `\vbox_unpack:N` and `\vbox_unpack_drop:N`. These functions are documented on page 318.)

```

\ vbox_set_split_to_ht:NNn Splitting a vertical box in two.
\ vbox_set_split_to_ht:cNn
\ vbox_set_split_to_ht:Ncn
\ vbox_set_split_to_ht:ccn
\ vbox_gset_split_to_ht:NNn
\ vbox_gset_split_to_ht:cNn
\ vbox_gset_split_to_ht:Ncn
\ vbox_gset_split_to_ht:ccn
36796 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
36797 { \tex_setbox:D #1 \tex_vsplit:D #2 to \_box_dim_eval:n {#3} }
36798 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
36799 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
36800 {
36801 \tex_global:D \tex_setbox:D #1
36802 \tex_vsplit:D #2 to \_box_dim_eval:n {#3}
36803 }
36804 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

```

(End of definition for `\vbox_set_split_to_ht:NNn` and `\vbox_gset_split_to_ht:NNn`. These functions are documented on page 318.)

97.12 Affine transformations

`\l__box_angle_fp` When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
36805 \fp_new:N \l__box_angle_fp
```

(End of definition for `\l__box_angle_fp`.)

`\l__box_cos_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\l__box_sin_fp
```

```
36806 \fp_new:N \l__box_cos_fp
36807 \fp_new:N \l__box_sin_fp
```

(End of definition for `\l__box_cos_fp` and `\l__box_sin_fp`.)

`\l__box_top_dim` These are the positions of the four edges of a box before manipulation.

```

\l__box_bottom_dim
\l__box_left_dim
\l__box_right_dim
36808 \dim_new:N \l__box_top_dim
36809 \dim_new:N \l__box_bottom_dim
36810 \dim_new:N \l__box_left_dim
36811 \dim_new:N \l__box_right_dim

```

(End of definition for `\l__box_top_dim` and others.)

`\l__box_top_new_dim` These are the positions of the four edges of a box after manipulation.

```

\l__box_bottom_new_dim
\l__box_left_new_dim
\l__box_right_new_dim
36812 \dim_new:N \l__box_top_new_dim
36813 \dim_new:N \l__box_bottom_new_dim
36814 \dim_new:N \l__box_left_new_dim
36815 \dim_new:N \l__box_right_new_dim

```

(End of definition for `\l__box_top_new_dim` and others.)

`\l__box_tmp_box` Scratch space, but also needed by some parts of the driver.

```
36816 \box_new:N \l__box_tmp_box
```

(End of definition for `\l__box_tmp_box`.)

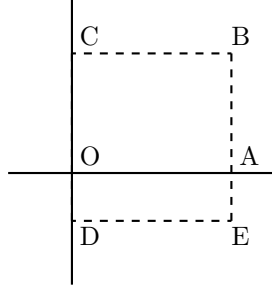


Figure 2: Coordinates of a box prior to rotation.

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual
\box_rotate:cn rotation is in an auxiliary to keep the flow slightly clearer

\box_grotate:Nn

\box_grotate:cn

__box_rotate:NnN

__box_rotate:N

__box_rotate_xdir:nnN

__box_rotate_ydir:nnN

__box_rotate_quadrant_one:

__box_rotate_quadrant_two:

__box_rotate_quadrant_three:

__box_rotate_quadrant_four:

```

36817 \cs_new_protected:Npn \box_rotate:Nn #1#2
36818 { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
36819 \cs_generate_variant:Nn \box_rotate:Nn { c }
36820 \cs_new_protected:Npn \box_grotate:Nn #1#2
36821 { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
36822 \cs_generate_variant:Nn \box_grotate:Nn { c }
36823 \cs_new_protected:Npn \__box_rotate:NnN #1#2#3
36824 {
36825   #3 #1
36826   {
36827     \fp_set:Nn \l__box_angle_fp {#2}
36828     \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
36829     \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
36830     \__box_rotate:N #1
36831   }
36832 }

```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```

36833 \cs_new_protected:Npn \__box_rotate:N #1
36834 {
36835   \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
36836   \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
36837   \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
36838   \dim_zero:N \l__box_left_dim

```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B , C , D and E is illustrated (Figure 2). The vertex O is the reference point on the baseline, and in this implementation is also the center of rotation. The formulae are, for a point P and angle α :

$$\begin{aligned}
 P'_x &= P_x - O_x \\
 P'_y &= P_y - O_y \\
 P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
 P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
 P'''_x &= P''_x + O_x + L_x \\
 P'''_y &= P''_y + O_y
 \end{aligned}$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x -coordinate 0. This is desirable as $\text{\TeX}$ boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```

36839 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
36840 {
36841     \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
36842     { \__box_rotate_quadrant_one: }
36843     { \__box_rotate_quadrant_two: }
36844 }
36845 {
36846     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
36847     { \__box_rotate_quadrant_three: }
36848     { \__box_rotate_quadrant_four: }
36849 }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current $\text{\TeX}$ reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

36850 \hbox_set:Nn \l__box_tmp_box { \box_use:N #1 }
36851 \hbox_set:Nn \l__box_tmp_box
36852 {
36853     \dim_horizontal:n { -\l__box_left_new_dim }
36854     \hbox:n
36855     {
36856         \__box_backend_rotate:Nn
36857         \l__box_tmp_box
36858         \l__box_angle_fp
36859     }
36860 }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

36861 \box_set_ht:Nn \l__box_tmp_box { \l__box_top_new_dim }
36862 \box_set_dp:Nn \l__box_tmp_box { -\l__box_bottom_new_dim }
36863 \box_set_wd:Nn \l__box_tmp_box
36864 { \l__box_right_new_dim - \l__box_left_new_dim }
36865 \box_use_drop:N \l__box_tmp_box
36866 }

```

These functions take a general point $(\#1,\#2)$ and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the `l3coffins` module, where both parts are needed.

```

36867 \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
36868 {
36869     \dim_set:Nn #3
36870     {
36871         \fp_to_dim:n
36872         {
36873             \l__box_cos_fp * \dim_to_fp:n {#1}
36874             - \l__box_sin_fp * \dim_to_fp:n {#2}

```

```

36875     }
36876   }
36877 }
36878 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
36879 {
36880   \dim_set:Nn #3
36881   {
36882     \fp_to_dim:n
36883     {
36884       \l__box_sin_fp * \dim_to_fp:n {#1}
36885       + \l__box_cos_fp * \dim_to_fp:n {#2}
36886     }
36887   }
36888 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y -values, whereas the left and right edges need the x -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

36889 \cs_new_protected:Npn \__box_rotate_quadrant_one:
36890 {
36891   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
36892   \l__box_top_new_dim
36893   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
36894   \l__box_bottom_new_dim
36895   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
36896   \l__box_left_new_dim
36897   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
36898   \l__box_right_new_dim
36899 }
36900 \cs_new_protected:Npn \__box_rotate_quadrant_two:
36901 {
36902   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
36903   \l__box_top_new_dim
36904   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
36905   \l__box_bottom_new_dim
36906   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
36907   \l__box_left_new_dim
36908   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
36909   \l__box_right_new_dim
36910 }
36911 \cs_new_protected:Npn \__box_rotate_quadrant_three:
36912 {
36913   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
36914   \l__box_top_new_dim
36915   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
36916   \l__box_bottom_new_dim
36917   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
36918   \l__box_left_new_dim
36919   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
36920   \l__box_right_new_dim
36921 }
36922 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

36923 {
36924   \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
36925   \l__box_top_new_dim
36926   \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
36927   \l__box_bottom_new_dim
36928   \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
36929   \l__box_left_new_dim
36930   \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
36931   \l__box_right_new_dim
36932 }

```

(End of definition for `\box_rotate:Nn` and others. These functions are documented on page 321.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.
`\l__box_scale_y_fp`

```

36933 \fp_new:N \l__box_scale_x_fp
36934 \fp_new:N \l__box_scale_y_fp

```

(End of definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize_to_wd_and_ht_plus_dp:cnm
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\__box_resize_to_wd_and_ht_plus_dp:NnnN
\__box_resize_set_corners:N
  \__box_resize:N
  \__box_resize:NNN
36935 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
36936 {
36937   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
36938   \hbox_set:Nn
36939 }
36940 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
36941 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
36942 {
36943   \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
36944   \hbox_gset:Nn
36945 }
36946 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
36947 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
36948 {
36949   #4 #1
36950   {
36951     \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

36952   \fp_set:Nn \l__box_scale_x_fp
36953   { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }

```

The y -scaling needs both the height and the depth of the current box.

```

36954   \fp_set:Nn \l__box_scale_y_fp
36955   {
36956     \dim_to_fp:n {#3}
36957     / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
36958   }

```

Hand off to the auxiliary which does the rest of the work.

```

36959   \__box_resize:N #1
36960 }
36961 }
36962 \cs_new_protected:Npn \__box_resize_set_corners:N #1
36963 {

```

```

36964 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
36965 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
36966 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
36967 \dim_zero:N \l__box_left_dim
36968 }

```

With at least one real scaling to do, the next phase is to find the new edge coordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

36969 \cs_new_protected:Npn \__box_resize:N #1
36970 {
36971   \__box_resize:NNN \l__box_right_new_dim
36972   \l__box_scale_x_fp \l__box_right_dim
36973   \__box_resize:NNN \l__box_bottom_new_dim
36974   \l__box_scale_y_fp \l__box_bottom_dim
36975   \__box_resize:NNN \l__box_top_new_dim
36976   \l__box_scale_y_fp \l__box_top_dim
36977   \__box_resize_common:N #1
36978 }
36979 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
36980 {
36981   \dim_set:Nn #1
36982   { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
36983 }

```

(End of definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 321.)

<pre> \box_resize_to_ht:Nn \box_resize_to_ht:cn \box_gresize_to_ht:Nn \box_gresize_to_ht:cn __box_resize_to_ht:NnN \box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:cn \box_gresize_to_ht_plus_dp:Nn \box_gresize_to_ht_plus_dp:cn __box_resize_to_ht_plus_dp:NnN \box_resize_to_wd:Nn \box_resize_to_wd:cn \box_gresize_to_wd:Nn \box_gresize_to_wd:cn __box_resize_to_wd:NnN \box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:cnn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:cnn __box_resize_to_wd_ht:NnnN </pre>	Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case). <pre> 36984 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2 36985 { __box_resize_to_ht:NnN #1 {#2} \hbox_set:Nn } 36986 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c } 36987 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2 36988 { __box_resize_to_ht:NnN #1 {#2} \hbox_gset:Nn } 36989 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c } 36990 \cs_new_protected:Npn __box_resize_to_ht:NnN #1#2#3 36991 { 36992 #3 #1 36993 { 36994 __box_resize_set_corners:N #1 36995 \fp_set:Nn \l__box_scale_y_fp 36996 { 36997 \dim_to_fp:n {#2} 36998 / \dim_to_fp:n { \l__box_top_dim } 36999 } 37000 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp 37001 __box_resize:N #1 37002 } 37003 } 37004 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2 </pre>

```

37005 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
37006 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
37007 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
37008 { \_box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
37009 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
37010 \cs_new_protected:Npn \_box_resize_to_ht_plus_dp:NnN #1#2#3
37011 {
37012   #3 #1
37013   {
37014     \_box_resize_set_corners:N #1
37015     \fp_set:Nn \l__box_scale_y_fp
37016     {
37017       \dim_to_fp:n {#2}
37018       / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
37019     }
37020     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
37021     \_box_resize:N #1
37022   }
37023 }
37024 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
37025 { \_box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
37026 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
37027 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
37028 { \_box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
37029 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
37030 \cs_new_protected:Npn \_box_resize_to_wd:NnN #1#2#3
37031 {
37032   #3 #1
37033   {
37034     \_box_resize_set_corners:N #1
37035     \fp_set:Nn \l__box_scale_x_fp
37036     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
37037     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
37038     \_box_resize:N #1
37039   }
37040 }
37041 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
37042 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
37043 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
37044 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
37045 { \_box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
37046 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
37047 \cs_new_protected:Npn \_box_resize_to_wd_and_ht:NnnN #1#2#3#4
37048 {
37049   #4 #1
37050   {
37051     \_box_resize_set_corners:N #1
37052     \fp_set:Nn \l__box_scale_x_fp
37053     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
37054     \fp_set:Nn \l__box_scale_y_fp
37055     {
37056       \dim_to_fp:n {#3}
37057       / \dim_to_fp:n { \l__box_top_dim }
37058     }

```



```

37059         \_box_resize:N #1
37060     }
37061 }

```

(End of definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 320.)

\box_scale:Nnn When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. **\box_scale:cnn** Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. **\box_gscale:Nnn** The code here is split into two as this allows sharing with the auto-resizing functions. **\box_gscale:cnn**

```

\__box_scale:NnnN
\__box_scale:N
37062 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
37063 { \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
37064 \cs_generate_variant:Nn \box_scale:Nnn { c }
37065 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
37066 { \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
37067 \cs_generate_variant:Nn \box_gscale:Nnn { c }
37068 \cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
37069 {
37070     #4 #1
37071     {
37072         \fp_set:Nn \l__box_scale_x_fp {#2}
37073         \fp_set:Nn \l__box_scale_y_fp {#3}
37074         \__box_scale:N #1
37075     }
37076 }
37077 \cs_new_protected:Npn \__box_scale:N #1
37078 {
37079     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
37080     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
37081     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
37082     \dim_zero:N \l__box_left_dim
37083     \dim_set:Nn \l__box_top_new_dim
37084     { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
37085     \dim_set:Nn \l__box_bottom_new_dim
37086     { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
37087     \dim_set:Nn \l__box_right_new_dim
37088     { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
37089     \__box_resize_common:N #1
37090 }

```

(End of definition for `\box_scale:Nnn` and others. These functions are documented on page 321.)

\box_autosize_to_wd_and_ht:Nnn Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere. **\box_autosize_to_wd_and_ht:cnn**

```

\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
37091 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
37092 { \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
37093 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
37094 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
37095 { \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
37096 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht:Nnn { c }
37097 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
37098 {
37099     \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

37100     \hbox_set:Nn
37101   }
37102   \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
37103   \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
37104   {
37105     \__box_autosize:NnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
37106     \hbox_gset:Nn
37107   }
37108   \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
37109   \cs_new_protected:Npn \__box_autosize:NnnN #1#2#3#4#5
37110   {
37111     #5 #1
37112     {
37113       \fp_set:Nn \l__box_scale_x_fp { ( \dim_to_fp:n {#2} ) / \box_wd:N #1 }
37114       \fp_set:Nn \l__box_scale_y_fp
37115       { ( \dim_to_fp:n {#3} ) / ( \dim_to_fp:n {#4} ) }
37116       \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
37117       { \fp_set_eq:Nn \l__box_scale_x_fp \l__box_scale_y_fp }
37118       { \fp_set_eq:Nn \l__box_scale_y_fp \l__box_scale_x_fp }
37119       \__box_scale:N #1
37120     }
37121   }

```

(End of definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 320.)

`\__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

37122   \cs_new_protected:Npn \__box_resize_common:N #1
37123   {
37124     \hbox_set:Nn \l__box_tmp_box
37125     {
37126       \__box_backend_scale:Nnn
37127       #1
37128       \l__box_scale_x_fp
37129       \l__box_scale_y_fp
37130     }

```

The new height and depth can be applied directly.

```

37131     \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
37132     {
37133       \box_set_ht:Nn \l__box_tmp_box { \l__box_top_new_dim }
37134       \box_set_dp:Nn \l__box_tmp_box { -\l__box_bottom_new_dim }
37135     }
37136     {
37137       \box_set_dp:Nn \l__box_tmp_box { \l__box_top_new_dim }
37138       \box_set_ht:Nn \l__box_tmp_box { -\l__box_bottom_new_dim }
37139     }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

37140     \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
37141     {

```

```

37142     \hbox_to_wd:nn { \l__box_right_new_dim }
37143     {
37144         \dim_horizontal:N \l__box_right_new_dim
37145         \box_use_drop:N \l__box_tmp_box
37146         \tex_hss:D
37147     }
37148 }
37149 {
37150     \box_set_wd:Nn \l__box_tmp_box { \l__box_right_new_dim }
37151     \hbox:n
37152     {
37153         \dim_horizontal:n{ Opt }
37154         \box_use_drop:N \l__box_tmp_box
37155         \tex_hss:D
37156     }
37157 }
37158 }

```

(End of definition for `\__box_resize_common:N`.)

97.13 Adding frames to boxes

`\__box_rule_horizontal:nn` Wrappers around the T_EX primitives for creating rules, with the usual dimension expression support. The non-expanding rule is set up in horizontal mode as that is usually the best plan. The `\scan_stop:` is needed here as T_EX continues to scan for keywords: multiple uses of the same word are permitted. (Thanks to Ruixi Zhang for pointing this out.) At some stage these will presumably move to a different (public) place.

```

37159 \cs_new_protected:Npn \__box_rule_horizontal:nn #1#2
37160 {
37161     \tex_hrulerule:D
37162     height \__box_dim_eval:n {#1}
37163     depth \__box_dim_eval:n {#2}
37164     \scan_stop:
37165 }
37166 \cs_new_protected:Npn \__box_rule_vertical:n #1
37167 { \tex_vrule:D width \__box_dim_eval:n {#1} \scan_stop: }

```

(End of definition for `\__box_rule_horizontal:nn` and `\__box_rule_vertical:n`.)

```

\box_frame:Nnn Framing a box requires the depth is known, so there has to be an assignment step first
\box_frame:cnn to allow this measurement. The actual insertion is done such that the content goes in
\box_gframe:Nnn “behind” the rules. That requires a bit of shuffling so that the final box is the correct
\box_gframe:cnn size, including the necessary border, while keeping the reference point in the right place.
__box_frame:nnNN
__box_frame:eeNN
37168 \cs_new_protected:Npn \box_frame:Nnn #1#2#3
37169 {
37170     \__box_frame:eeNN { \dim_eval:n {#2} } { \dim_eval:n {#3} }
37171     #1 \hbox_set:Nn
37172 }
37173 \cs_generate_variant:Nn \box_frame:Nnn { c }
37174 \cs_new_protected:Npn \box_gframe:Nnn #1#2#3
37175 {
37176     \__box_frame:eeNN { \dim_eval:n {#2} } { \dim_eval:n {#3} }
37177     #1 \hbox_gset:Nn

```

```

37178 }
37179 \cs_generate_variant:Nn \box_gframe:Nnn { c }
37180 \cs_new_protected:Npn \__box_frame:nnNN #1#2#3#4
37181 {
37182   #4 #3
37183   {
37184     \dim_horizontal:n { #1 + #2 }
37185     \box_use:N #3
37186     \dim_horizontal:n { -\box_wd:N #3 - #1 - #2 }
37187     \box_move_down:nn { \box_dp:N #3 + #1 + #2 }
37188     {
37189       \vbox:n
37190       {
37191         \__box_rule_horizontal:nn {#2} { \c_zero_dim }
37192         \hbox:n
37193         {
37194           \__box_rule_vertical:n {#2}
37195           \dim_horizontal:n { #1 * 2 + \box_wd:N #3 }
37196           \vbox:n
37197           {
37198             \dim_vertical:n
37199             { #1 * 2 + \box_dp:N #3 + \box_ht:N #3 }
37200           }
37201           \__box_rule_vertical:n {#2}
37202         }
37203         \__box_rule_horizontal:nn {#2} { \c_zero_dim }
37204       }
37205     }
37206   }
37207 }
37208 \cs_generate_variant:Nn \__box_frame:nnNN { ee }

```

(End of definition for `\box_frame:Nnn`, `\box_gframe:Nnn`, and `\__box_frame:nnNN`. These functions are documented on page 322.)

```

\box_underline:Nnn A cut-down version of framing, with just one side marked.
\box_underline:cnn
\box_gunderline:Nnn
\box_gunderline:cnn
\__box_underline:nnNN
\__box_underline:eeNN
37209 \cs_new_protected:Npn \box_underline:Nnn #1#2#3
37210 {
37211   \__box_underline:eeNN
37212   { \dim_eval:n {#2} } { \dim_eval:n {#3} }
37213   #1 \hbox_set:Nn
37214 }
37215 \cs_generate_variant:Nn \box_underline:Nnn { c }
37216 \cs_new_protected:Npn \box_gunderline:Nnn #1#2#3
37217 {
37218   \__box_underline:eeNN
37219   { \dim_eval:n {#2} } { \dim_eval:n {#3} }
37220   #1 \hbox_gset:Nn
37221 }
37222 \cs_generate_variant:Nn \box_gunderline:Nnn { c }
37223 \cs_new_protected:Npn \__box_underline:nnNN #1#2#3#4
37224 {
37225   #4 #3
37226   {

```

```

37227 \box_use:N #3
37228 \dim_horizontal:n { -\box_wd:N #3 }
37229 \box_move_down:nn { \box_dp:N #3 + #1 + #2 }
37230 {
37231 \vbox:n
37232 {
37233 \hbox:n { \dim_horizontal:n { \box_wd:N #3 } }
37234 \__box_rule_horizontal:nn {#2} { \c_zero_dim }
37235 }
37236 }
37237 }
37238 }
37239 \cs_generate_variant:Nn \__box_underline:nnNN { ee }

```

(End of definition for `\box_underline:Nnn`, `\box_gunderline:Nnn`, and `\__box_underline:nnNN`. These functions are documented on page 322.)

97.14 Viewing part of a box

`\box_set_clipped:N`
`\box_set_clipped:c`
`\box_gset_clipped:N`
`\box_gset_clipped:c`

A wrapper around the driver-dependent code.

```

37240 \cs_new_protected:Npn \box_set_clipped:N #1
37241 { \hbox_set:Nn #1 { \__box_backend_clip:N #1 } }
37242 \cs_generate_variant:Nn \box_set_clipped:N { c }
37243 \cs_new_protected:Npn \box_gset_clipped:N #1
37244 { \hbox_gset:Nn #1 { \__box_backend_clip:N #1 } }
37245 \cs_generate_variant:Nn \box_gset_clipped:N { c }

```

(End of definition for `\box_set_clipped:N` and `\box_gset_clipped:N`. These functions are documented on page 322.)

`\box_set_trim:Nnnnn`
`\box_set_trim:cnnnn`
`\box_gset_trim:Nnnnn`
`\box_gset_trim:cnnnn`
`\__box_set_trim:NnnnnN`

Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```

37246 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
37247 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
37248 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
37249 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
37250 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
37251 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
37252 \cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
37253 {
37254 \hbox_set:Nn \l__box_tmp_box
37255 {
37256 \dim_horizontal:n { -#2 }
37257 \box_use:N #1
37258 \dim_horizontal:n { -#4 }
37259 }

```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

37260 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
37261 {
37262   \hbox_set:Nn \l__box_tmp_box
37263   {
37264     \box_move_down:nn \c_zero_dim
37265     { \box_use_drop:N \l__box_tmp_box }
37266   }
37267   \box_set_dp:Nn \l__box_tmp_box { \box_dp:N #1 - (#3) }
37268 }
37269 {
37270   \hbox_set:Nn \l__box_tmp_box
37271   {
37272     \box_move_down:nn { (#3) - \box_dp:N #1 }
37273     { \box_use_drop:N \l__box_tmp_box }
37274   }
37275   \box_set_dp:Nn \l__box_tmp_box \c_zero_dim
37276 }

```

Same thing, this time from the top of the box.

```

37277 \dim_compare:nNnTF { \box_ht:N \l__box_tmp_box } > {#5}
37278 {
37279   \hbox_set:Nn \l__box_tmp_box
37280   {
37281     \box_move_up:nn \c_zero_dim
37282     { \box_use_drop:N \l__box_tmp_box }
37283   }
37284   \box_set_ht:Nn \l__box_tmp_box
37285   { \box_ht:N \l__box_tmp_box - (#5) }
37286 }
37287 {
37288   \hbox_set:Nn \l__box_tmp_box
37289   {
37290     \box_move_up:nn { (#5) - \box_ht:N \l__box_tmp_box }
37291     { \box_use_drop:N \l__box_tmp_box }
37292   }
37293   \box_set_ht:Nn \l__box_tmp_box \c_zero_dim
37294 }
37295 #6 #1 \l__box_tmp_box
37296 }

```

(End of definition for `\box_set_trim:Nnnnn`, `\box_gset_trim:Nnnnn`, and `\__box_set_trim:NnnnnN`. These functions are documented on page 322.)

`\box_set_viewport:Nnnnn`
`\box_set_viewport:cnnnn`
`\box_gset_viewport:Nnnnn`
`\box_gset_viewport:cnnnn`
`\__box_viewport:NnnnnN`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

37297 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
37298 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
37299 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
37300 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
37301 { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
37302 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
37303 \cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6
37304 {
37305   \hbox_set:Nn \l__box_tmp_box
37306   {

```

```

37307     \dim_horizontal:n { -#2 }
37308     \box_use:N #1
37309     \dim_horizontal:n { #4 - \box_wd:N #1 }
37310   }
37311 \dim_compare:nNnTF {#3} < \c_zero_dim
37312 {
37313   \hbox_set:Nn \l__box_tmp_box
37314   {
37315     \box_move_down:nn \c_zero_dim
37316     { \box_use_drop:N \l__box_tmp_box }
37317   }
37318   \box_set_dp:Nn \l__box_tmp_box { - \__box_dim_eval:n {#3} }
37319 }
37320 {
37321   \hbox_set:Nn \l__box_tmp_box
37322   { \box_move_down:nn {#3} { \box_use_drop:N \l__box_tmp_box } }
37323   \box_set_dp:Nn \l__box_tmp_box \c_zero_dim
37324 }
37325 \dim_compare:nNnTF {#5} > \c_zero_dim
37326 {
37327   \hbox_set:Nn \l__box_tmp_box
37328   {
37329     \box_move_up:nn \c_zero_dim
37330     { \box_use_drop:N \l__box_tmp_box }
37331   }
37332   \box_set_ht:Nn \l__box_tmp_box
37333   {
37334     (#5)
37335     \dim_compare:nNnT {#3} > \c_zero_dim
37336     { - (#3) }
37337   }
37338 }
37339 {
37340   \hbox_set:Nn \l__box_tmp_box
37341   {
37342     \box_move_up:nn { - \__box_dim_eval:n {#5} }
37343     { \box_use_drop:N \l__box_tmp_box }
37344   }
37345   \box_set_ht:Nn \l__box_tmp_box \c_zero_dim
37346 }
37347 #6 #1 \l__box_tmp_box
37348 }

```

(End of definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `\__box_viewport:NnnnnN`.
These functions are documented on page 323.)

37349 `\code`

Chapter 98

l3coffins implementation

```
37350 <*code>
37351 <@@=coffin>
```

98.1 Coffins: data structures and general variables

```
\l__coffin_tmp_box Scratch variables.
\l__coffin_tmp_dim
\l__coffin_tmp_tl
37352 \box_new:N \l__coffin_tmp_box
37353 \dim_new:N \l__coffin_tmp_dim
37354 \tl_new:N \l__coffin_tmp_tl

(End of definition for \l__coffin_tmp_box, \l__coffin_tmp_dim, and \l__coffin_tmp_tl.)

\c__coffin_corners_prop The “corners”; of a coffin define the real content, as opposed to the TeX bounding box.
They all start off in the same place, of course.
37355 \prop_const_from_keyval:Nn \c__coffin_corners_prop
37356 {
37357     tl = { Opt } { Opt } ,
37358     tr = { Opt } { Opt } ,
37359     bl = { Opt } { Opt } ,
37360     br = { Opt } { Opt } ,
37361 }

(End of definition for \c__coffin_corners_prop.)

\c__coffin_poles_prop Pole positions are given for horizontal, vertical and reference-point based values.
37362 \prop_const_from_keyval:Nn \c__coffin_poles_prop
37363 {
37364     l  = { Opt } { Opt } { Opt } { 1000pt } ,
37365     hc = { Opt } { Opt } { Opt } { 1000pt } ,
37366     r  = { Opt } { Opt } { Opt } { 1000pt } ,
37367     b  = { Opt } { Opt } { 1000pt } { Opt } ,
37368     vc = { Opt } { Opt } { 1000pt } { Opt } ,
37369     t  = { Opt } { Opt } { 1000pt } { Opt } ,
37370     B  = { Opt } { Opt } { 1000pt } { Opt } ,
37371     H  = { Opt } { Opt } { 1000pt } { Opt } ,
37372     T  = { Opt } { Opt } { 1000pt } { Opt } ,
37373 }
```


(End of definition for `\c__coffin_poles_prop`.)

`\l__coffin_slope_A_fp` Used for calculations of intersections.
`\l__coffin_slope_B_fp` 37374 `\fp_new:N \l__coffin_slope_A_fp`
37375 `\fp_new:N \l__coffin_slope_B_fp`
(End of definition for `\l__coffin_slope_A_fp` and `\l__coffin_slope_B_fp`.)

`\l__coffin_error_bool` For propagating errors so that parts of the code can work around them.
37376 `\bool_new:N \l__coffin_error_bool`
(End of definition for `\l__coffin_error_bool`.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected
`\l__coffin_offset_y_dim` from those requested in an alignment for the positions of the handles.
37377 `\dim_new:N \l__coffin_offset_x_dim`
37378 `\dim_new:N \l__coffin_offset_y_dim`
(End of definition for `\l__coffin_offset_x_dim` and `\l__coffin_offset_y_dim`.)

`\l__coffin_x_dim` For calculating intersections and so forth.
`\l__coffin_y_dim` 37379 `\dim_new:N \l__coffin_x_dim`
`\l__coffin_x_prime_dim` 37380 `\dim_new:N \l__coffin_y_dim`
`\l__coffin_y_prime_dim` 37381 `\dim_new:N \l__coffin_x_prime_dim`
37382 `\dim_new:N \l__coffin_y_prime_dim`
(End of definition for `\l__coffin_x_dim` and others.)

98.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`\__coffin_to_value:N` Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

37383 `\cs_new_eq:NN \__coffin_to_value:N \tex_number:D`

(End of definition for `\__coffin_to_value:N`.)

`\coffin_if_exist_p:N` Several of the higher-level coffin functions would give multiple errors if the coffin does
`\coffin_if_exist_p:c` not exist. A cleaner way to handle this is provided here: both the box and the coffin
`\coffin_if_exist:N $\underline{T}$  $\underline{F}$` structure are checked.

`\coffin_if_exist:c $\underline{T}$  $\underline{F}$` 37384 `\prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }`
37385 `{`
37386 `\cs_if_exist:N $\underline{T}$  $\underline{F}$  #1`
37387 `{`
37388 `\cs_if_exist:c $\underline{T}$  $\underline{F}$  { coffin ~ \__coffin_to_value:N #1 ~ poles }`
37389 `{ \prg_return_true: }`
37390 `{ \prg_return_false: }`
37391 `}`
37392 `{ \prg_return_false: }`
37393 `}`
37394 `\prg_generate_conditional_variant:Nnn \coffin_if_exist:N`
37395 `{ c } { p , T , F , TF }`

(End of definition for `\coffin_if_exist:NTF`. This function is documented on page 325.)

`\__coffin_if_exist:NT` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

37396 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
37397 {
37398   \coffin_if_exist:NTF #1
37399   { #2 }
37400   {
37401     \msg_error:nne { coffin } { unknown }
37402     { \token_to_str:N #1 }
37403   }
37404 }
```

(End of definition for `\__coffin_if_exist:NT`.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c 37405 \cs_new_protected:Npn \coffin_clear:N #1
\coffin_gclear:N 37406 {
\coffin_gclear:c 37407   \__coffin_if_exist:NT #1
37408   {
37409     \box_clear:N #1
37410     \__coffin_reset_structure:N #1
37411   }
37412 }
37413 \cs_generate_variant:Nn \coffin_clear:N { c }
37414 \cs_new_protected:Npn \coffin_gclear:N #1
37415 {
37416   \__coffin_if_exist:NT #1
37417   {
37418     \box_gclear:N #1
37419     \__coffin_greset_structure:N #1
37420   }
37421 }
37422 \cs_generate_variant:Nn \coffin_gclear:N { c }
```

(End of definition for `\coffin_clear:N` and `\coffin_gclear:N`. These functions are documented on page 325.)

`\coffin_new:N` Creating a new coffin means making the underlying box and adding the data structures. The `\debug_suspend:` and `\debug_resume:` functions prevent `\prop_gclear_new:c` from writing useless information to the log file.

```

37423 \cs_new_protected:Npn \coffin_new:N #1
37424 {
37425   \box_new:N #1
37426   \debug_suspend:
37427   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
37428   \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
37429   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
37430   \c__coffin_corners_prop
37431   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
37432   \c__coffin_poles_prop
37433   \debug_resume:
```

```

37434 }
37435 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End of definition for `\coffin_new:N`. This function is documented on page 325.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
`\hcoffin_set:cn` update the handle positions.

```

\hcoffin_gset:Nn 37436 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
\hcoffin_gset:cn 37437 {
37438   \__coffin_if_exist:NT #1
37439   {
37440     \hbox_set:Nn #1
37441     {
37442       \color_ensure_current:
37443       #2
37444     }
37445     \coffin_reset_poles:N #1
37446   }
37447 }
37448 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
37449 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
37450 {
37451   \__coffin_if_exist:NT #1
37452   {
37453     \hbox_gset:Nn #1
37454     {
37455       \color_ensure_current:
37456       #2
37457     }
37458     \coffin_greset_poles:N #1
37459   }
37460 }
37461 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End of definition for `\hcoffin_set:Nn` and `\hcoffin_gset:Nn`. These functions are documented on page 326.)

`\vcoffin_set:Nnn` Setting vertical coffins is more complex. First, the material is typeset with a given width.
`\vcoffin_set:cnn` The default handles and poles are set as for a horizontal coffin, before finding the top
`\vcoffin_gset:Nnn` baseline using a temporary box. No `\color_ensure_current:` here as that would add a
`\vcoffin_gset:cnn` whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

\__coffin_set_vertical:NnnNNN 37462 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
\__coffin_set_vertical_aux: 37463 {
37464   \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
37465   \vbox_set:Nn \coffin_reset_poles:N \__coffin_set_pole:Nnn
37466 }
37467 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
37468 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
37469 {
37470   \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
37471   \vbox_gset:Nn \coffin_greset_poles:N \__coffin_gset_pole:Nnn
37472 }
37473 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }

```

```

37474 \cs_new_protected:Npn \__coffin_set_vertical:NnnNNN #1#2#3#4#5#6
37475 {
37476   \__coffin_if_exist:NT #1
37477   {
37478     #4 #1
37479     {
37480       \dim_set:Nn \tex_hsize:D {#2}
37481       \__coffin_set_vertical_aux:
37482       #3
37483     }
37484     #5 #1
37485     \vbox_set_top:Nn \l__coffin_tmp_box { \vbox_unpack:N #1 }
37486     #6 #1 { T }
37487     {
37488       { Opt }
37489       {
37490         \dim_eval:n
37491         { \box_ht:N #1 - \box_ht:N \l__coffin_tmp_box }
37492       }
37493       { 1000pt }
37494       { Opt }
37495     }
37496     \box_clear:N \l__coffin_tmp_box
37497   }
37498 }
37499 \cs_new_protected:Npe \__coffin_set_vertical_aux:
37500 {
37501   \bool_lazy_and:nnT
37502   { \cs_if_exist_p:N \fmtname }
37503   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
37504   {
37505     \dim_set_eq:NN \exp_not:N \linewidth \tex_hsize:D
37506     \dim_set_eq:NN \exp_not:N \columnwidth \tex_hsize:D
37507   }
37508 }

```

(End of definition for \vcoffin_set:Nnn and others. These functions are documented on page 326.)

\hcoffin_set:Nw These are the “begin”/“end” versions of the above: watch the grouping!

```

\hcoffin_set:cw 37509 \cs_new_protected:Npn \hcoffin_set:Nw #1
\hcoffin_gset:Nw 37510 {
\hcoffin_gset:cw 37511   \__coffin_if_exist:NT #1
\hcoffin_set_end: 37512   {
\hcoffin_gset_end: 37513     \hbox_set:Nw #1 \color_ensure_current:
37514     \cs_set_protected:Npn \hcoffin_set_end:
37515     {
37516       \hbox_set_end:
37517       \coffin_reset_poles:N #1
37518     }
37519   }
37520 }
37521 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
37522 \cs_new_protected:Npn \hcoffin_gset:Nw #1
37523 {

```

```

37524 \__coffin_if_exist:NT #1
37525 {
37526     \hbox_gset:Nw #1 \color_ensure_current:
37527     \cs_set_protected:Npn \hcoffin_gset_end:
37528     {
37529         \hbox_gset_end:
37530         \coffin_greset_poles:N #1
37531     }
37532 }
37533 }
37534 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
37535 \cs_new_protected:Npn \hcoffin_set_end: { }
37536 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End of definition for \hcoffin_set:Nw and others. These functions are documented on page 326.)

```

\vcoffin_set:Nnw The same for vertical coffins.
\vcoffin_set:cnw 37537 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
\vcoffin_gset:Nnw 37538 {
\vcoffin_gset:cnw 37539 \__coffin_set_vertical:NnNNNNNw #1 {#2} \vbox_set:Nw
\__coffin_set_vertical:NnNNNNNw 37540 \vcoffin_set_end:
\vcoffin_set_end: 37541 \vbox_set_end: \coffin_reset_poles:N \__coffin_set_pole:Nnn
\vcoffin_gset_end: 37542 }
37543 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
37544 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
37545 {
37546     \__coffin_set_vertical:NnNNNNNw #1 {#2} \vbox_gset:Nw
37547     \vcoffin_gset_end:
37548     \vbox_gset_end: \coffin_greset_poles:N \__coffin_gset_pole:Nnn
37549 }
37550 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
37551 \cs_new_protected:Npn \__coffin_set_vertical:NnNNNNNw #1#2#3#4#5#6#7
37552 {
37553     \__coffin_if_exist:NT #1
37554     {
37555         #3 #1
37556         \dim_set:Nn \tex_hsize:D {#2}
37557         \__coffin_set_vertical_aux:
37558         \cs_set_protected:Npn #4
37559         {
37560             #5
37561             #6 #1
37562             \vbox_set_top:Nn \l__coffin_tmp_box { \vbox_unpack:N #1 }
37563             #7 #1 { T }
37564             {
37565                 { Opt }
37566                 {
37567                     \dim_eval:n
37568                     { \box_ht:N #1 - \box_ht:N \l__coffin_tmp_box }
37569                 }
37570                 { 1000pt }
37571                 { Opt }
37572             }
37573             \box_clear:N \l__coffin_tmp_box

```

```

37574     }
37575   }
37576 }
37577 \cs_new_protected:Npn \vcoffin_set_end: { }
37578 \cs_new_protected:Npn \vcoffin_gset_end: { }

```

(End of definition for \vcoffin_set:Nnw and others. These functions are documented on page 326.)

```

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.
\coffin_set_eq:Nc 37579 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 37580 {
\coffin_set_eq:cc 37581   \__coffin_if_exist:NT #2
\coffin_gset_eq:NN 37582   {
\coffin_gset_eq:Nc 37583     \box_set_eq:NN #1 #2
\coffin_gset_eq:cN 37584     \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
\coffin_gset_eq:cc 37585     { coffin ~ \__coffin_to_value:N #2 ~ corners }
37586     \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
37587     { coffin ~ \__coffin_to_value:N #2 ~ poles }
37588   }
37589 }
37590 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
37591 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
37592 {
37593   \__coffin_if_exist:NT #2
37594   {
37595     \box_gset_eq:NN #1 #2
37596     \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
37597     { coffin ~ \__coffin_to_value:N #2 ~ corners }
37598     \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
37599     { coffin ~ \__coffin_to_value:N #2 ~ poles }
37600   }
37601 }
37602 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End of definition for \coffin_set_eq:NN and \coffin_gset_eq:NN. These functions are documented on page 325.)

\c_empty_coffin Special coffins: these cannot be set up earlier as they need \coffin_new:N. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

37603 \coffin_new:N \c_empty_coffin
37604 \coffin_new:N \l__coffin_aligned_coffin
37605 \coffin_new:N \l__coffin_aligned_internal_coffin

```

(End of definition for \c_empty_coffin, \l__coffin_aligned_coffin, and \l__coffin_aligned_internal_coffin. This variable is documented on page 330.)

\l_tmpa_coffin The usual scratch space.

```

\l_tmpb_coffin 37606 \coffin_new:N \l_tmpa_coffin
\g_tmpa_coffin 37607 \coffin_new:N \l_tmpb_coffin
\g_tmpb_coffin 37608 \coffin_new:N \g_tmpa_coffin
37609 \coffin_new:N \g_tmpb_coffin

```

(End of definition for \l_tmpa_coffin and others. These variables are documented on page 330.)

98.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

`\coffin_dp:c`

`\coffin_ht:N`

`\coffin_ht:c`

`\coffin_ht_plus_dp:N`

`\coffin_ht_plus_dp:c`

`\coffin_wd:N`

`\coffin_wd:c`

```

37610 \cs_new_eq:NN \coffin_dp:N \box_dp:N
37611 \cs_new_eq:NN \coffin_dp:c \box_dp:c
37612 \cs_new_eq:NN \coffin_ht:N \box_ht:N
37613 \cs_new_eq:NN \coffin_ht:c \box_ht:c
37614 \cs_new_eq:NN \coffin_ht_plus_dp:N \box_ht_plus_dp:N
37615 \cs_new_eq:NN \coffin_ht_plus_dp:c \box_ht_plus_dp:c
37616 \cs_new_eq:NN \coffin_wd:N \box_wd:N
37617 \cs_new_eq:NN \coffin_wd:c \box_wd:c

```

(End of definition for `\coffin_dp:N` and others. These functions are documented on page 328.)

98.4 Coffins: handle and pole management

`\coffin_pole:Nn` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

`\__coffin_pole:nNn`

```

37618 \cs_new:Npn \coffin_pole:Nn #1#2
37619 {
37620   \exp_args:Ne \__coffin_pole:nNn
37621   { \prop_item:cn { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} }
37622   #1 {#2}
37623 }
37624 \cs_new:Npn \__coffin_pole:nNn #1#2#3
37625 {
37626   \tl_if_blank:nTF {#1}
37627   {
37628     \msg_expandable_error:nnee { coffin } { unknown-pole }
37629     { \exp_not:n {#3} } { \token_to_str:N #2 }
37630     { Opt } { Opt } { Opt } { Opt }
37631   }
37632   {#1}
37633 }

```

(End of definition for `\coffin_pole:Nn` and `\__coffin_pole:nNn`. This function is documented on page 327.)

`\__coffin_reset_structure:N` Resetting the structure is a simple copy job.

`\__coffin_greset_structure:N`

```

37634 \cs_new_protected:Npn \__coffin_reset_structure:N #1
37635 {
37636   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
37637   \c__coffin_corners_prop
37638   \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
37639   \c__coffin_poles_prop
37640 }
37641 \cs_new_protected:Npn \__coffin_greset_structure:N #1
37642 {
37643   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
37644   \c__coffin_corners_prop
37645   \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
37646   \c__coffin_poles_prop
37647 }

```

(End of definition for `\__coffin_reset_structure:N` and `\__coffin_greset_structure:N`.)

`\coffin_set_horizontal_pole:Nnn` Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

`\coffin_set_horizontal_pole:cnm`

`\coffin_gset_horizontal_pole:Nnn`

`\coffin_gset_horizontal_pole:cnm`

`\__coffin_set_horizontal_pole:NnnN`

`\coffin_set_vertical_pole:Nnn`

`\coffin_set_vertical_pole:cnm`

`\coffin_gset_vertical_pole:Nnn`

`\coffin_gset_vertical_pole:cnm`

`\__coffin_set_vertical_pole:NnnN`

`\__coffin_set_pole:Nnn`

`\__coffin_gset_pole:Nnn`

```

37648 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
37649 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cne }
37650 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
37651 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
37652 { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
37653 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
37654 \cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
37655 {
37656   \__coffin_if_exist:NT #1
37657   {
37658     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
37659     {#2}
37660     {
37661       { Opt } { \dim_eval:n {#3} }
37662       { 1000pt } { Opt }
37663     }
37664   }
37665 }
37666 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
37667 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cne }
37668 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
37669 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
37670 { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
37671 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
37672 \cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
37673 {
37674   \__coffin_if_exist:NT #1
37675   {
37676     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
37677     {#2}
37678     {
37679       { \dim_eval:n {#3} } { Opt }
37680       { Opt } { 1000pt }
37681     }
37682   }
37683 }
37684 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
37685 {
37686   \prop_put:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
37687   {#2} {#3}
37688 }
37689 \cs_new_protected:Npn \__coffin_gset_pole:Nnn #1#2#3
37690 {
37691   \prop_gput:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
37692   {#2} {#3}
37693 }
```

(End of definition for `\coffin_set_horizontal_pole:Nnn` and others. These functions are documented on page 326.)

`\coffin_reset_poles:N` Simple shortcuts.

```

\coffin_greset_poles:N
37694 \cs_new_protected:Npn \coffin_reset_poles:N #1
37695 {
37696   \__coffin_reset_structure:N #1
37697   \__coffin_update_corners:N #1
37698   \__coffin_update_poles:N #1
37699 }
37700 \cs_new_protected:Npn \coffin_greset_poles:N #1
37701 {
37702   \__coffin_greset_structure:N #1
37703   \__coffin_gupdate_corners:N #1
37704   \__coffin_gupdate_poles:N #1
37705 }

```

(End of definition for `\coffin_reset_poles:N` and `\coffin_greset_poles:N`. These functions are documented on page 327.)

`\__coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying $\text{\TeX}$ box.

```

\__coffin_gupdate_corners:N
\__coffin_update_corners:NN
\__coffin_update_corners:NNN
37706 \cs_new_protected:Npn \__coffin_update_corners:N #1
37707 { \__coffin_update_corners:NN #1 \prop_put:Nne }
37708 \cs_new_protected:Npn \__coffin_gupdate_corners:N #1
37709 { \__coffin_update_corners:NN #1 \prop_gput:Nne }
37710 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
37711 {
37712   \exp_args:Nc \__coffin_update_corners:NNN
37713   { coffin ~ \__coffin_to_value:N #1 ~ corners }
37714   #1 #2
37715 }
37716 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
37717 {
37718   #3 #1
37719   { tl }
37720   { { Opt } { \dim_eval:n { \box_ht:N #2 } } }
37721   #3 #1
37722   { tr }
37723   {
37724     { \dim_eval:n { \box_wd:N #2 } }
37725     { \dim_eval:n { \box_ht:N #2 } }
37726   }
37727   #3 #1
37728   { bl }
37729   { { Opt } { \dim_eval:n { -\box_dp:N #2 } } }
37730   #3 #1
37731   { br }
37732   {
37733     { \dim_eval:n { \box_wd:N #2 } }
37734     { \dim_eval:n { -\box_dp:N #2 } }
37735   }
37736 }

```

(End of definition for `\__coffin_update_corners:N` and others.)

`\__coffin_update_poles:N` This function is called when a coffin is set, and updates the poles to reflect the nature
`\__coffin_gupdate_poles:N` of size of the box. Thus this function only alters poles where the default position is
`\__coffin_update_poles:NN`
`\__coffin_update_poles:NNN`

dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

37737 \cs_new_protected:Npn \__coffin_update_poles:N #1
37738 { \__coffin_update_poles:NN #1 \prop_put:Nne }
37739 \cs_new_protected:Npn \__coffin_gupdate_poles:N #1
37740 { \__coffin_update_poles:NN #1 \prop_gput:Nne }
37741 \cs_new_protected:Npn \__coffin_update_poles:NN #1#2
37742 {
37743   \exp_args:Nc \__coffin_update_poles:NNN
37744     { coffin ~ \__coffin_to_value:N #1 ~ poles }
37745     #1 #2
37746 }
37747 \cs_new_protected:Npn \__coffin_update_poles:NNN #1#2#3
37748 {
37749   #3 #1 { hc }
37750   {
37751     { \dim_eval:n { 0.5 \box_wd:N #2 } }
37752     { Opt } { Opt } { 1000pt }
37753   }
37754   #3 #1 { r }
37755   {
37756     { \dim_eval:n { \box_wd:N #2 } }
37757     { Opt } { Opt } { 1000pt }
37758   }
37759   #3 #1 { vc }
37760   {
37761     { Opt }
37762     { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
37763     { 1000pt }
37764     { Opt }
37765   }
37766   #3 #1 { t }
37767   {
37768     { Opt }
37769     { \dim_eval:n { \box_ht:N #2 } }
37770     { 1000pt }
37771     { Opt }
37772   }
37773   #3 #1 { b }
37774   {
37775     { Opt }
37776     { \dim_eval:n { -\box_dp:N #2 } }
37777     { 1000pt }
37778     { Opt }
37779   }
37780 }

```

(End of definition for `\__coffin_update_poles:N` and others.)

98.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which

```

\__coffin_calculate_intersection:Nnn
\__coffin_calculate_intersection:nnnnnnnn
\__coffin_calculate_intersection:nnnnnnn

```

an error trap is needed.

```

37781 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
37782 {
37783   \bool_set_false:N \l__coffin_error_bool
37784   \use:e
37785   {
37786     \__coffin_calculate_intersection:nnnnnnnn
37787     \coffin_pole:Nn #1 {#2}
37788     \coffin_pole:Nn #1 {#3}
37789   }
37790   \bool_if:NT \l__coffin_error_bool
37791   {
37792     \msg_error:nn { coffin } { no-pole-intersection }
37793     \dim_zero:N \l__coffin_x_dim
37794     \dim_zero:N \l__coffin_y_dim
37795   }
37796 }

```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d') . These are arguments 1–4 and 5–8, respectively. In both cases a and b are the coordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c' . However, if one of the poles is either horizontal or vertical then one or more of c , d , c' and d' are zero and a special case is needed.

```

37797 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
37798 #1#2#3#4#5#6#7#8
37799 {
37800   \dim_compare:nNnTF {#3} = \c_zero_dim

```

The case where the first pole is vertical. So the x -component of the interaction is at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

37801 {
37802   \dim_set:Nn \l__coffin_x_dim {#1}
37803   \dim_compare:nNnTF {#7} = \c_zero_dim
37804   { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the y -component of the intersection is b' . If not,

$$y = \frac{d'}{c'}(a - a') + b'$$

with the x -component already known to be $\#1$.

```

37805 {
37806   \dim_set:Nn \l__coffin_y_dim
37807   {
37808     \dim_compare:nNnTF {#8} = \c_zero_dim
37809     {#6}
37810     {
37811       \fp_to_dim:n
37812       {
37813         ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
37814         * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
37815         + \dim_to_fp:n {#6}
37816       }
37817     }

```

```

37818     }
37819   }
37820 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x - and y -components interchanged.

```

37821 {
37822   \dim_compare:nNnTF {#4} = \c_zero_dim
37823   {
37824     \dim_set:Nn \l__coffin_y_dim {#2}
37825     \dim_compare:nNnTF {#8} = { \c_zero_dim }
37826     { \bool_set_true:N \l__coffin_error_bool }
37827   }

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'}(b - b') + a'$$

which is again handled by the same auxiliary.

```

37828   \dim_set:Nn \l__coffin_x_dim
37829   {
37830     \dim_compare:nNnTF {#7} = \c_zero_dim
37831     {#5}
37832     {
37833       \fp_to_dim:n
37834       {
37835         ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
37836         * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
37837         + \dim_to_fp:n {#5}
37838       }
37839     }
37840   }
37841 }
37842 }

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

37843 {
37844   \use:e
37845   {
37846     \__coffin_calculate_intersection:nnnnnn
37847     { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
37848     { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
37849   }
37850   {#1} {#2} {#5} {#6}
37851 }
37852 }
37853 }

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the x -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the y -value with

$$y = s(x - a) + b$$

```

37854 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
37855 {
37856   \fp_compare:nNnTF {#1} = {#2}
37857   { \bool_set_true:N \l__coffin_error_bool }
37858   {
37859     \dim_set:Nn \l__coffin_x_dim
37860     {
37861       \fp_to_dim:n
37862       {
37863         (
37864           #1 * \dim_to_fp:n {#3}
37865           - #2 * \dim_to_fp:n {#5}
37866           - \dim_to_fp:n {#4}
37867           + \dim_to_fp:n {#6}
37868         )
37869         /
37870         ( #1 - #2 )
37871       }
37872     }
37873     \dim_set:Nn \l__coffin_y_dim
37874     {
37875       \fp_to_dim:n
37876       {
37877         #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
37878         + \dim_to_fp:n {#4}
37879       }
37880     }
37881   }
37882 }

```

(End of definition for `\__coffin_calculate_intersection:Nnn`, `\__coffin_calculate_intersection:nnnnnnnn`, and `\__coffin_calculate_intersection:nnnnnn`.)

98.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.
`\l__coffin_cos_fp`

```

37883 \fp_new:N \l__coffin_sin_fp
37884 \fp_new:N \l__coffin_cos_fp

```

(End of definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

37885 \prop_new:N \l__coffin_bounding_prop

```

(End of definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.
`\l__coffin_poles_prop`

```

37886 \prop_new:N \l__coffin_corners_prop
37887 \prop_new:N \l__coffin_poles_prop

```

(End of definition for \l__coffin_corners_prop and \l__coffin_poles_prop.)

\l__coffin_bounding_shift_dim The shift of the bounding box of a coffin from the real content.

```
37888 \dim_new:N \l__coffin_bounding_shift_dim
```

(End of definition for \l__coffin_bounding_shift_dim.)

\l__coffin_left_corner_dim \l__coffin_right_corner_dim \l__coffin_bottom_corner_dim \l__coffin_top_corner_dim These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
37889 \dim_new:N \l__coffin_left_corner_dim
```

```
37890 \dim_new:N \l__coffin_right_corner_dim
```

```
37891 \dim_new:N \l__coffin_bottom_corner_dim
```

```
37892 \dim_new:N \l__coffin_top_corner_dim
```

(End of definition for \l__coffin_left_corner_dim and others.)

\coffin_rotate:Nn Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set \l__coffin_sin_fp and \l__coffin_cos_fp, which are carried through unchanged for the rest of the procedure.

\coffin_rotate:cn

\coffin_grotate:Nn

\coffin_grotate:cn

__coffin_rotate:NnNNN

```
37893 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
```

```
37894 { \__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cn \hbox_set:Nn }
```

```
37895 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

```
37896 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
```

```
37897 { \__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cn \hbox_gset:Nn }
```

```
37898 \cs_generate_variant:Nn \coffin_grotate:Nn { c }
```

```
37899 \cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5
```

```
37900 {
```

```
37901   \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
```

```
37902   \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
37903   \prop_set_eq:Nc \l__coffin_corners_prop
```

```
37904   { coffin ~ \__coffin_to_value:N #1 ~ corners }
```

```
37905   \prop_set_eq:Nc \l__coffin_poles_prop
```

```
37906   { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
37907   \prop_map_inline:Nn \l__coffin_corners_prop
```

```
37908   { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
```

```
37909   \prop_map_inline:Nn \l__coffin_poles_prop
```

```
37910   { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
37911   \__coffin_set_bounding:N #1
```

```
37912   \prop_map_inline:Nn \l__coffin_bounding_prop
```

```
37913   { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
37914   \__coffin_find_corner_maxima:N #1
```

```
37915   \__coffin_find_bounding_shift:
```

```
37916   #3 #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

37917     \hbox_set:Nn \l__coffin_tmp_box
37918     {
37919         \dim_horizontal:n
37920         { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
37921         \box_move_down:nn { \l__coffin_bottom_corner_dim }
37922         { \box_use:N #1 }
37923     }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

37924     \box_set_ht:Nn \l__coffin_tmp_box
37925     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
37926     \box_set_dp:Nn \l__coffin_tmp_box { Opt }
37927     \box_set_wd:Nn \l__coffin_tmp_box
37928     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
37929     #5 #1 { \box_use_drop:N \l__coffin_tmp_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

37930     \prop_map_inline:Nn \l__coffin_corners_prop
37931     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
37932     \prop_map_inline:Nn \l__coffin_poles_prop
37933     { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

37934     #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
37935     \l__coffin_corners_prop
37936     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
37937     \l__coffin_poles_prop
37938 }

```

(End of definition for \coffin_rotate:Nn, \coffin_grotate:Nn, and __coffin_rotate:NnNNN. These functions are documented on page 327.)

`\__coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

37939     \cs_new_protected:Npn \__coffin_set_bounding:N #1
37940     {
37941         \prop_put:Nne \l__coffin_bounding_prop { tl }
37942         { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
37943         \prop_put:Nne \l__coffin_bounding_prop { tr }
37944         {
37945             { \dim_eval:n { \box_wd:N #1 } }
37946             { \dim_eval:n { \box_ht:N #1 } }
37947         }

```

```

37948 \dim_set:Nn \l__coffin_tmp_dim { -\box_dp:N #1 }
37949 \prop_put:Nne \l__coffin_bounding_prop { bl }
37950 { { Opt } { \dim_use:N \l__coffin_tmp_dim } }
37951 \prop_put:Nne \l__coffin_bounding_prop { br }
37952 {
37953   { \dim_eval:n { \box_wd:N #1 } }
37954   { \dim_use:N \l__coffin_tmp_dim }
37955 }
37956 }

```

(End of definition for __coffin_set_bounding:N.)

__coffin_rotate_bounding:nnn

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

__coffin_rotate_corner:Nnnn

```

37957 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
37958 {
37959   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
37960   \prop_put:Nne \l__coffin_bounding_prop {#1}
37961   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
37962 }
37963 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
37964 {
37965   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
37966   \prop_put:Nne \l__coffin_corners_prop {#2}
37967   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
37968 }

```

(End of definition for __coffin_rotate_bounding:nnn and __coffin_rotate_corner:Nnnn.)

__coffin_rotate_pole:Nnnnnn

Rotating a single pole simply means shifting the coordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

37969 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
37970 {
37971   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
37972   \__coffin_rotate_vector:nnNN {#5} {#6}
37973   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
37974   \prop_put:Nne \l__coffin_poles_prop {#2}
37975   {
37976     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
37977     { \dim_use:N \l__coffin_x_prime_dim }
37978     { \dim_use:N \l__coffin_y_prime_dim }
37979   }
37980 }

```

(End of definition for __coffin_rotate_pole:Nnnnnn.)

__coffin_rotate_vector:nnNN

A rotation function, which needs only an input vector (as dimensions) and an output space. The values \l__coffin_cos_fp and \l__coffin_sin_fp should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

37981 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
37982 {
37983   \dim_set:Nn #3

```



```

37984     {
37985         \fp_to_dim:n
37986         {
37987             \dim_to_fp:n {#1} * \l__coffin_cos_fp
37988             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
37989         }
37990     }
37991     \dim_set:Nn #4
37992     {
37993         \fp_to_dim:n
37994         {
37995             \dim_to_fp:n {#1} * \l__coffin_sin_fp
37996             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
37997         }
37998     }
37999 }

```

(End of definition for __coffin_rotate_vector:nnNN.)

__coffin_find_corner_maxima:N
__coffin_find_corner_maxima_aux:nn

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

38000 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
38001 {
38002     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
38003     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
38004     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
38005     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
38006     \prop_map_inline:Nn \l__coffin_corners_prop
38007     { \__coffin_find_corner_maxima_aux:nn ##2 }
38008 }
38009 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
38010 {
38011     \dim_set:Nn \l__coffin_left_corner_dim
38012     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
38013     \dim_set:Nn \l__coffin_right_corner_dim
38014     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
38015     \dim_set:Nn \l__coffin_bottom_corner_dim
38016     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
38017     \dim_set:Nn \l__coffin_top_corner_dim
38018     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
38019 }

```

(End of definition for __coffin_find_corner_maxima:N and __coffin_find_corner_maxima_aux:nn.)

__coffin_find_bounding_shift:
__coffin_find_bounding_shift_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

38020 \cs_new_protected:Npn \__coffin_find_bounding_shift:
38021 {
38022     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
38023     \prop_map_inline:Nn \l__coffin_bounding_prop
38024     { \__coffin_find_bounding_shift_aux:nn ##2 }

```

```

38025     }
38026 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
38027 {
38028     \dim_set:Nn \l__coffin_bounding_shift_dim
38029     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
38030 }

```

(End of definition for __coffin_find_bounding_shift: and __coffin_find_bounding_shift_aux:nn.)

__coffin_shift_corner:Nnnn Shifting the corners and poles of a coffin means subtracting the appropriate values from the x - and y -components. For the poles, this means that the direction vector is unchanged.

__coffin_shift_pole:Nnnnnn

```

38031 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
38032 {
38033     \prop_put:Nne \l__coffin_corners_prop {#2}
38034     {
38035         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
38036         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
38037     }
38038 }
38039 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
38040 {
38041     \prop_put:Nne \l__coffin_poles_prop {#2}
38042     {
38043         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
38044         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
38045         {#5} {#6}
38046     }
38047 }

```

(End of definition for __coffin_shift_corner:Nnnn and __coffin_shift_pole:Nnnnnn.)

\l__coffin_scale_x_fp Storage for the scaling factors in x and y , respectively.

\l__coffin_scale_y_fp

```

38048 \fp_new:N \l__coffin_scale_x_fp
38049 \fp_new:N \l__coffin_scale_y_fp

```

(End of definition for \l__coffin_scale_x_fp and \l__coffin_scale_y_fp.)

\l__coffin_scaled_total_height_dim When scaling, the values given have to be turned into absolute values.

\l__coffin_scaled_width_dim

```

38050 \dim_new:N \l__coffin_scaled_total_height_dim
38051 \dim_new:N \l__coffin_scaled_width_dim

```

(End of definition for \l__coffin_scaled_total_height_dim and \l__coffin_scaled_width_dim.)

\coffin_resize:Nnn Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

\coffin_resize:cnn

\coffin_gresize:Nnn

\coffin_gresize:cnn

__coffin_resize:NnnNN

```

38052 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
38053 {
38054     \__coffin_resize:NnnNN #1 {#2} {#3}
38055     \box_resize_to_wd_and_ht_plus_dp:Nnn
38056     \prop_set_eq:cN
38057 }

```

```

38058 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
38059 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
38060 {
38061   \__coffin_resize:NnnNN #1 {#2} {#3}
38062   \box_gresize_to_wd_and_ht_plus_dp:Nnn
38063   \prop_gset_eq:cN
38064 }
38065 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
38066 \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
38067 {
38068   \fp_set:Nn \l__coffin_scale_x_fp
38069   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
38070   \fp_set:Nn \l__coffin_scale_y_fp
38071   {
38072     \dim_to_fp:n {#3}
38073     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
38074   }
38075   #4 #1 {#2} {#3}
38076   \__coffin_resize_common:NnnN #1 {#2} {#3} #5
38077 }

```

(End of definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `\__coffin_resize:NnnNN`. These functions are documented on page 327.)

`\__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

38078 \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
38079 {
38080   \prop_set_eq:Nc \l__coffin_corners_prop
38081   { coffin ~ \__coffin_to_value:N #1 ~ corners }
38082   \prop_set_eq:Nc \l__coffin_poles_prop
38083   { coffin ~ \__coffin_to_value:N #1 ~ poles }
38084   \prop_map_inline:Nn \l__coffin_corners_prop
38085   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
38086   \prop_map_inline:Nn \l__coffin_poles_prop
38087   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values place the poles in the wrong location: this is corrected here.

```

38088   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
38089   {
38090     \prop_map_inline:Nn \l__coffin_corners_prop
38091     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
38092     \prop_map_inline:Nn \l__coffin_poles_prop
38093     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
38094   }
38095   #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
38096   \l__coffin_corners_prop
38097   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
38098   \l__coffin_poles_prop
38099 }

```

(End of definition for `\__coffin_resize_common:NnnN`.)

```

\coffin_scale:Nnn
\coffin_scale:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn
\__coffin_scale:NnnNN

```

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The

scaling is done the T_EX way as this works properly with floating point values without needing to use the fp module.

```

38100 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
38101 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
38102 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
38103 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
38104 { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
38105 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
38106 \cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
38107 {
38108   \fp_set:Nn \l__coffin_scale_x_fp {#2}
38109   \fp_set:Nn \l__coffin_scale_y_fp {#3}
38110   #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
38111   \dim_set:Nn \l__coffin_tmp_dim
38112     { \coffin_ht:N #1 + \coffin_dp:N #1 }
38113   \dim_set:Nn \l__coffin_scaled_total_height_dim
38114     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_tmp_dim }
38115   \dim_set:Nn \l__coffin_scaled_width_dim
38116     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
38117   \__coffin_resize_common:NnnN #1
38118     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
38119   #5
38120 }

```

(End of definition for `\coffin_scale:Nnn`, `\coffin_gscale:Nnn`, and `\__coffin_scale:NnnNN`. These functions are documented on page 327.)

`\__coffin_scale_vector:nnNN`

This function scales a vector from the origin using the pre-set scale factors in x and y . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

38121 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
38122 {
38123   \dim_set:Nn #3
38124     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
38125   \dim_set:Nn #4
38126     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
38127 }

```

(End of definition for `\__coffin_scale_vector:nnNN`.)

`\__coffin_scale_corner:Nnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

`\__coffin_scale_pole:Nnnnnn`

```

38128 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
38129 {
38130   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
38131   \prop_put:Nne \l__coffin_corners_prop {#2}
38132     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
38133 }
38134 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
38135 {
38136   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
38137   \prop_put:Nne \l__coffin_poles_prop {#2}
38138   {
38139     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
38140     {#5} {#6}

```

```

38141     }
38142 }

```

(End of definition for `\_coffin_scale_corner:Nnnn` and `\_coffin_scale_pole:Nnnnnn`.)

```

\_coffin_x_shift_corner:Nnnn
\_coffin_x_shift_pole:Nnnnnn

```

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

38143 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
38144 {
38145   \prop_put:Nne \l__coffin_corners_prop {#2}
38146   {
38147     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
38148   }
38149 }
38150 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
38151 {
38152   \prop_put:Nne \l__coffin_poles_prop {#2}
38153   {
38154     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
38155     {#5} {#6}
38156   }
38157 }

```

(End of definition for `\_coffin_x_shift_corner:Nnnn` and `\_coffin_x_shift_pole:Nnnnnn`.)

98.7 Aligning and typesetting of coffins

```

\coffin_join:NnnNnnnn
\coffin_join:cnnNnnnn
\coffin_join:Nnnncnnnn
\coffin_join:cnncnnnn

```

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

\coffin_gjoin:NnnNnnnn 38158 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
\coffin_gjoin:cnnNnnnn 38159 {
\coffin_gjoin:Nnnncnnnn 38160   \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
\coffin_gjoin:cnncnnnn 38161   \coffin_set_eq:NN
\_coffin_join:NnnNnnnnN 38162 }
38163 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
38164 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
38165 {
38166   \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
38167   \coffin_gset_eq:NN
38168 }
38169 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
38170 \cs_new_protected:Npn \_coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
38171 {
38172   \_coffin_align:NnnNnnnnN
38173   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin

```

Correct the placement of the reference point. If the x -offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x -offset and the width of the second box. So a second kern may be needed.

```

38174   \hbox_set:Nn \l__coffin_aligned_coffin

```

```

38175     {
38176         \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
38177         { \dim_horizontal:n { -\l__coffin_offset_x_dim } }
38178         \hbox_unpack:N \l__coffin_aligned_coffin
38179         \dim_set:Nn \l__coffin_tmp_dim
38180         { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
38181         \dim_compare:nNnT \l__coffin_tmp_dim < \c_zero_dim
38182         { \dim_horizontal:n { -\l__coffin_tmp_dim } }
38183     }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

38184     \__coffin_reset_structure:N \l__coffin_aligned_coffin
38185     \prop_clear:c
38186     {
38187         coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
38188         \c_space_tl corners
38189     }
38190     \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

38191     \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
38192     {
38193         \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
38194         \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
38195         \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
38196         \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
38197     }
38198     {
38199         \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
38200         \__coffin_offset_poles:Nnn #4
38201         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
38202         \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
38203         \__coffin_offset_corners:Nnn #4
38204         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
38205     }
38206     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
38207     #9 #1 \l__coffin_aligned_coffin
38208 }

```

(End of definition for \coffin_join:NnnNnnnn, \coffin_gjoin:NnnNnnnn, and __coffin_join:NnnNnnnnN. These functions are documented on page 328.)

<pre> \coffin_attach:NnnNnnnn \coffin_attach:cnnNnnnn \coffin_attach:NnnNnnnn \coffin_attach:cnnNnnnn \coffin_gattach:NnnNnnnn \coffin_gattach:cnnNnnnn \coffin_gattach:NnnNnnnn \coffin_gattach:cnnNnnnn __coffin_attach:NnnNnnnnN __coffin_attach_mark:NnnNnnnn </pre>	A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates. <pre> 38209 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8 38210 { 38211 __coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} 38212 \coffin_set_eq:NN 38213 } 38214 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnc } </pre>

```

38215 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
38216 {
38217   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
38218   \coffin_gset_eq:NN
38219 }
38220 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
38221 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
38222 {
38223   \__coffin_align:NnnNnnnnN
38224   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
38225   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
38226   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
38227   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
38228   \__coffin_reset_structure:N \l__coffin_aligned_coffin
38229   \prop_set_eq:cc
38230   {
38231     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
38232     \c_space_tl corners
38233   }
38234   { coffin ~ \__coffin_to_value:N #1 ~ corners }
38235   \__coffin_update_poles:N \l__coffin_aligned_coffin
38236   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
38237   \__coffin_offset_poles:Nnn #4
38238   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
38239   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
38240   #9 #1 \l__coffin_aligned_coffin
38241 }
38242 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
38243 {
38244   \__coffin_align:NnnNnnnnN
38245   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
38246   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
38247   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
38248   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
38249   \box_set_eq:NN #1 \l__coffin_aligned_coffin
38250 }

```

(End of definition for \coffin_attach:NnnNnnnn and others. These functions are documented on page 328.)

__coffin_align:NnnNnnnnN

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

38251 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
38252 {
38253   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
38254   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
38255   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
38256   \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
38257   \dim_set:Nn \l__coffin_offset_x_dim

```

```

38258     { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
38259 \dim_set:Nn \l__coffin_offset_y_dim
38260     { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
38261 \hbox_set:Nn \l__coffin_aligned_internal_coffin
38262     {
38263       \box_use:N #1
38264       \dim_horizontal:n { -\box_wd:N #1 }
38265       \dim_horizontal:n { \l__coffin_offset_x_dim }
38266       \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
38267     }
38268 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
38269 }

```

(End of definition for __coffin_align:NnnNnnnnN.)

__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnnn

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping over the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labeled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

38270 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
38271 {
38272   \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
38273   { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
38274 }
38275 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
38276 {
38277   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
38278   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
38279   \tl_if_in:nnTF {#2} { - }
38280   { \tl_set:Nn \l__coffin_tmp_tl { {#2} } }
38281   { \tl_set:Nn \l__coffin_tmp_tl { { #1 - #2 } } }
38282   \exp_last_unbraced:NNo \__coffin_set_pole:Nnn \l__coffin_aligned_coffin
38283   { \l__coffin_tmp_tl }
38284   {
38285     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
38286     {#5} {#6}
38287   }
38288 }

```

(End of definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

__coffin_offset_corners:Nnn
 __coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

38289 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
38290 {
38291   \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
38292   { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
38293 }
38294 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
38295 {
38296   \prop_put:cne

```



```

38297     {
38298         coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
38299         \c_space_tl corners
38300     }
38301     { #1 - #2 }
38302     {
38303         { \dim_eval:n { #3 + #5 } }
38304         { \dim_eval:n { #4 + #6 } }
38305     }
38306 }

```

(End of definition for __coffin_offset_corners:Nnn and __coffin_offset_corner:Nnnnn.)

```

\__coffin_update_vertical_poles:NNN
\__coffin_update_T:nnnnnnnnN
\__coffin_update_B:nnnnnnnnN

```

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

38307 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
38308 {
38309     \use:e
38310     {
38311         \__coffin_update_T:nnnnnnnnN
38312         \coffin_pole:Nn #3 { #1 -T }
38313         \coffin_pole:Nn #3 { #2 -T }
38314         #3
38315     }
38316     \use:e
38317     {
38318         \__coffin_update_B:nnnnnnnnN
38319         \coffin_pole:Nn #3 { #1 -B }
38320         \coffin_pole:Nn #3 { #2 -B }
38321         #3
38322     }
38323 }
38324 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
38325 {
38326     \dim_compare:nNnTF {#2} < {#6}
38327     {
38328         \__coffin_set_pole:Nnn #9 { T }
38329         { { Opt } {#6} { 1000pt } { Opt } }
38330     }
38331     {
38332         \__coffin_set_pole:Nnn #9 { T }
38333         { { Opt } {#2} { 1000pt } { Opt } }
38334     }
38335 }
38336 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
38337 {
38338     \dim_compare:nNnTF {#2} < {#6}
38339     {
38340         \__coffin_set_pole:Nnn #9 { B }
38341         { { Opt } {#2} { 1000pt } { Opt } }
38342     }
38343     {
38344         \__coffin_set_pole:Nnn #9 { B }

```

```

38345         { { Opt } {#6} { 1000pt } { Opt } }
38346     }
38347 }

```

(End of definition for `\_coffin_update_vertical_poles:NNN`, `\_coffin_update_T:nnnnnnnnN`, and `\_coffin_update_B:nnnnnnnnN`.)

`\c__coffin_empty_coffin` An empty-but-horizontal coffin.

```

38348 \coffin_new:N \c__coffin_empty_coffin
38349 \tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

```

(End of definition for `\c__coffin_empty_coffin`.)

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```

38350 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
38351 {
38352     \mode_leave_vertical:
38353     \_coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { 1 }
38354     #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
38355     \box_use_drop:N \l__coffin_aligned_coffin
38356 }
38357 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

```

(End of definition for `\coffin_typeset:Nnnnn`. This function is documented on page 328.)

98.8 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

```

\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin
38358 \coffin_new:N \l__coffin_display_coffin
38359 \coffin_new:N \l__coffin_display_coord_coffin
38360 \coffin_new:N \l__coffin_display_pole_coffin

```

(End of definition for `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, and `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```

38361 \prop_new:N \l__coffin_display_handles_prop
38362 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
38363 { { b } { r } { -1 } { 1 } }
38364 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
38365 { { b } { hc } { 0 } { 1 } }
38366 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
38367 { { b } { l } { 1 } { 1 } }
38368 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
38369 { { vc } { r } { -1 } { 0 } }
38370 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
38371 { { vc } { hc } { 0 } { 0 } }
38372 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
38373 { { vc } { l } { 1 } { 0 } }
38374 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
38375 { { t } { r } { -1 } { -1 } }

```

```

38376 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
38377   { { t } { hc } { 0 } { -1 } }
38378 \prop_put:Nnn \l__coffin_display_handles_prop { br }
38379   { { t } { l } { 1 } { -1 } }
38380 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
38381   { { t } { r } { -1 } { -1 } }
38382 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
38383   { { t } { hc } { 0 } { -1 } }
38384 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
38385   { { t } { l } { 1 } { -1 } }
38386 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
38387   { { vc } { r } { -1 } { 1 } }
38388 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
38389   { { vc } { hc } { 0 } { 1 } }
38390 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
38391   { { vc } { l } { 1 } { 1 } }
38392 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
38393   { { b } { r } { -1 } { -1 } }
38394 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
38395   { { b } { hc } { 0 } { -1 } }
38396 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
38397   { { b } { l } { 1 } { -1 } }

```

(End of definition for \l__coffin_display_handles_prop.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

38398 \dim_new:N \l__coffin_display_offset_dim
38399 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End of definition for \l__coffin_display_offset_dim.)

`\l__coffin_display_x_dim` As the intersections of poles have to be calculated to find which ones to print, there is
`\l__coffin_display_y_dim` a need to avoid repetition. This is done by saving the intersection into two dedicated
values.

```

38400 \dim_new:N \l__coffin_display_x_dim
38401 \dim_new:N \l__coffin_display_y_dim

```

(End of definition for \l__coffin_display_x_dim and \l__coffin_display_y_dim.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a
“nice” output.

```

38402 \prop_new:N \l__coffin_display_poles_prop

```

(End of definition for \l__coffin_display_poles_prop.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

38403 \tl_new:N \l__coffin_display_font_tl
38404 \bool_lazy_and:nnT
38405   { \cs_if_exist_p:N \fmtname }
38406   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
38407   {
38408     \tl_set:Nn \l__coffin_display_font_tl
38409       { \sffamily \tiny }
38410   }

```

(End of definition for \l__coffin_display_font_tl.)

__coffin_rule:nn Abstract out creation of rules here until there is a higher-level interface.

```

38411 \cs_new_protected:Npn \__coffin_rule:nn #1#2
38412 {
38413   \mode_leave_vertical:
38414   \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: }
38415 }

```

(End of definition for __coffin_rule:nn.)

\coffin_mark_handle:Nnnn Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimized version for showing all handles which comes next.

\coffin_mark_handle:cnnn
 __coffin_mark_handle_aux:nnnnNnn

```

38416 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
38417 {
38418   \hcoffin_set:Nn \l__coffin_display_pole_coffin
38419   {
38420     \color_select:n {#4}
38421     \__coffin_rule:nn { 1pt } { 1pt }
38422   }
38423   \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
38424   \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
38425   \hcoffin_set:Nn \l__coffin_display_coord_coffin
38426   {
38427     \color_select:n {#4}
38428     \l__coffin_display_font_tl
38429     ( \tl_to_str:n { #2 , #3 } )
38430   }
38431   \prop_get:NnN \l__coffin_display_handles_prop
38432   { #2 #3 } \l__coffin_tmp_tl
38433   \quark_if_no_value:NTF \l__coffin_tmp_tl
38434   {
38435     \prop_get:NnN \l__coffin_display_handles_prop
38436     { #3 #2 } \l__coffin_tmp_tl
38437     \quark_if_no_value:NTF \l__coffin_tmp_tl
38438     {
38439       \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
38440       \l__coffin_display_coord_coffin { l } { vc }
38441       { 1pt } { Opt }
38442     }
38443     {
38444       \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
38445       \l__coffin_tmp_tl #1 {#2} {#3}
38446     }
38447   }
38448   {
38449     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
38450     \l__coffin_tmp_tl #1 {#2} {#3}
38451   }
38452 }
38453 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
38454 {

```

```

38455     \__coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
38456     \l__coffin_display_coord_coffin {#1} {#2}
38457     { #3 \l__coffin_display_offset_dim }
38458     { #4 \l__coffin_display_offset_dim }
38459 }
38460 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End of definition for `\coffin_mark_handle:Nnnn` and `\__coffin_mark_handle_aux:nnnnNnn`. This function is documented on page 329.)

```

\coffin_display_handles:Nn
\coffin_display_handles:cn
  \__coffin_display_handles_aux:nnnnnn
  \__coffin_display_handles_aux:nnnn
  \__coffin_display_attach:Nnnnn

```

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

38461 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
38462 {
38463   \hcoffin_set:Nn \l__coffin_display_pole_coffin
38464   {
38465     \color_select:n {#2}
38466     \__coffin_rule:nn { 1pt } { 1pt }
38467   }
38468   \prop_set_eq:Nc \l__coffin_display_poles_prop
38469   { coffin ~ \__coffin_to_value:N #1 ~ poles }
38470   \str_if_eq:eeT
38471   { \coffin_pole:Nn #1 { H } }
38472   { \coffin_pole:Nn #1 { T } }
38473   { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
38474   \str_if_eq:eeT
38475   { \coffin_pole:Nn #1 { H } }
38476   { \coffin_pole:Nn #1 { B } }
38477   { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
38478   \coffin_set_eq:NN \l__coffin_display_coffin #1
38479   \prop_map_inline:Nn \l__coffin_display_poles_prop
38480   {
38481     \prop_remove:Nn \l__coffin_display_poles_prop {##1}
38482     \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
38483   }
38484   \box_use_drop:N \l__coffin_display_coffin
38485 }

```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```

38486 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
38487 {
38488   \prop_map_inline:Nn \l__coffin_display_poles_prop
38489   {
38490     \bool_set_false:N \l__coffin_error_bool
38491     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
38492     \bool_if:NF \l__coffin_error_bool
38493     {
38494       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
38495       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
38496       \__coffin_display_attach:Nnnnn

```

```

38497         \l__coffin_display_pole_coffin { hc } { vc }
38498     { Opt } { Opt }
38499     \hcoffin_set:Nn \l__coffin_display_coord_coffin
38500     {
38501         \color_select:n {#6}
38502         \l__coffin_display_font_tl
38503         ( \tl_to_str:n { #1 , ##1 } )
38504     }
38505     \prop_get:NnN \l__coffin_display_handles_prop
38506     { #1 ##1 } \l__coffin_tmp_tl
38507     \quark_if_no_value:NTF \l__coffin_tmp_tl
38508     {
38509         \prop_get:NnN \l__coffin_display_handles_prop
38510         { ##1 #1 } \l__coffin_tmp_tl
38511         \quark_if_no_value:NTF \l__coffin_tmp_tl
38512         {
38513             \__coffin_display_attach:Nnnnn
38514             \l__coffin_display_coord_coffin { l } { vc }
38515             { 1pt } { Opt }
38516         }
38517         {
38518             \exp_last_unbraced:No
38519             \__coffin_display_handles_aux:nnnn
38520             \l__coffin_tmp_tl
38521         }
38522     }
38523     {
38524         \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
38525         \l__coffin_tmp_tl
38526     }
38527 }
38528 }
38529 }
38530 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
38531 {
38532     \__coffin_display_attach:Nnnnn
38533     \l__coffin_display_coord_coffin {#1} {#2}
38534     { #3 \l__coffin_display_offset_dim }
38535     { #4 \l__coffin_display_offset_dim }
38536 }
38537 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

```

This is a dedicated version of \coffin_attach:NnnNnnnn with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```

38538 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
38539 {
38540     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
38541     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
38542     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
38543     \dim_set:Nn \l__coffin_offset_x_dim
38544     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
38545     \dim_set:Nn \l__coffin_offset_y_dim
38546     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }

```

```

38547 \hbox_set:Nn \l__coffin_aligned_coffin
38548 {
38549   \box_use:N \l__coffin_display_coffin
38550   \dim_horizontal:n { -\box_wd:N \l__coffin_display_coffin }
38551   \dim_horizontal:n { \l__coffin_offset_x_dim }
38552   \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
38553 }
38554 \box_set_ht:Nn \l__coffin_aligned_coffin
38555 { \box_ht:N \l__coffin_display_coffin }
38556 \box_set_dp:Nn \l__coffin_aligned_coffin
38557 { \box_dp:N \l__coffin_display_coffin }
38558 \box_set_wd:Nn \l__coffin_aligned_coffin
38559 { \box_wd:N \l__coffin_display_coffin }
38560 \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
38561 }

```

(End of definition for `\coffin_display_handles:Nn` and others. This function is documented on page 329.)

`\coffin_show_structure:N` For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
\__coffin_show_structure:NN
38562 \cs_new_protected:Npn \coffin_show_structure:N
38563 { \__coffin_show_structure:NN \msg_show:nneeee }
38564 \cs_generate_variant:Nn \coffin_show_structure:N { c }
38565 \cs_new_protected:Npn \coffin_log_structure:N
38566 { \__coffin_show_structure:NN \msg_log:nneeee }
38567 \cs_generate_variant:Nn \coffin_log_structure:N { c }
38568 \cs_new_protected:Npn \__coffin_show_structure:NN #1#2
38569 {
38570   \__coffin_if_exist:NT #2
38571   {
38572     #1 { coffin } { show }
38573     { \token_to_str:N #2 }
38574     {
38575       \iow_newline:>~ ht ~~~ \dim_eval:n { \coffin_ht:N #2 }
38576       \iow_newline:>~ dp ~~~ \dim_eval:n { \coffin_dp:N #2 }
38577       \iow_newline:>~ wd ~~~ \dim_eval:n { \coffin_wd:N #2 }
38578     }
38579     {
38580       \prop_map_function:cN
38581       { coffin ~ \__coffin_to_value:N #2 ~ poles }
38582       \msg_show_item_unbraced:nn
38583     }
38584     { }
38585   }
38586 }

```

(End of definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `\__coffin_show_structure:NN`. These functions are documented on page 329.)

`\coffin_show:N` Essentially a combination of `\coffin_show_structure:N` and `\box_show:Nnn`, but we need to avoid having two prompts, so we use `\msg_term:nneeee` instead of `\msg_show:nneeee` in the show case.

```

\coffin_log:N
\coffin_log:c
\coffin_show:Nnn
38587 \cs_new_protected:Npn \coffin_show:N #1
38588 { \coffin_show:Nnn #1 \c_max_int \c_max_int }

```

```

\coffin_show:cnn
\coffin_log:Nnn
\coffin_log:cnn
\__coffin_show:NNNnn

```

```

38589 \cs_generate_variant:Nn \coffin_show:N { c }
38590 \cs_new_protected:Npn \coffin_log:N #1
38591 { \coffin_log:Nnn #1 \c_max_int \c_max_int }
38592 \cs_generate_variant:Nn \coffin_log:N { c }
38593 \cs_new_protected:Npn \coffin_show:Nnn
38594 { \__coffin_show:NNNnn \msg_term:nneeee \box_show:Nnn }
38595 \cs_generate_variant:Nn \coffin_show:Nnn { c }
38596 \cs_new_protected:Npn \coffin_log:Nnn
38597 { \__coffin_show:NNNnn \msg_log:nneeee \box_show:Nnn }
38598 \cs_generate_variant:Nn \coffin_log:Nnn { c }
38599 \cs_new_protected:Npn \__coffin_show:NNNnn #1#2#3#4#5
38600 {
38601   \__coffin_if_exist:NT #3
38602   {
38603     \__coffin_show_structure:NN #1 #3
38604     #2 #3 {#4} {#5}
38605   }
38606 }

```

(End of definition for `\coffin_show:N` and others. These functions are documented on page [329](#).)

98.9 Messages

```

38607 \msg_new:nnnn { coffin } { no-pole-intersection }
38608 { No~intersection~between~coffin~poles. }
38609 {
38610   LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
38611   but~they~do~not~have~a~unique~meeting~point:~
38612   the~value~(Opt,~Opt)~will~be~used.
38613 }
38614 \msg_new:nnnn { coffin } { unknown }
38615 { Unknown~coffin~'#1'. }
38616 { The~coffin~'#1'~was~never~defined. }
38617 \msg_new:nnn { coffin } { unknown-pole }
38618 { Pole~'#1'~unknown~for~coffin~'#2'. }
38619 \msg_new:nnn { coffin } { show }
38620 {
38621   Size~of~coffin~#1 : #2 \\
38622   Poles~of~coffin~#1 : #3 .
38623 }
38624 </code>

```


Chapter 99

l3color implementation

```
38625 <*code>
38626 <@@=color>
```

99.1 Basics

`\l__color_current_tl` The color currently active for foreground (text, etc.) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- `gray` `<gray>` Grayscale color with the `<gray>` value running from 0 (fully black) to 1 (fully white)
- `cmYk` `<cyan>` `<magenta>` `<yellow>` `<black>`
- `rgb` `<red>` `<green>` `<blue>`

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- `spot` `<name>` `<tint>` A pre-defined spot color, where the `<name>` should be a pre-defined string color name and the `<tint>` should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

TeXhackers note: The content of `\l__color_current_tl` comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(End of definition for \l__color_current_tl.)

`\color_group_begin:` Grouping for color is the same as using the basic `\group_begin:` and `\group_end:` functions. However, for semantic reasons, they are renamed here.

```
38627 \cs_new_eq:NN \color_group_begin: \group_begin:
38628 \cs_new_eq:NN \color_group_end: \group_end:
```

(End of definition for \color_group_begin: and \color_group_end:. These functions are documented on page 331.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color “now”.

```
38629 \cs_new_protected:Npn \color_ensure_current:
38630 { \__color_select:N \l__color_current_tl }
```

(End of definition for \color_ensure_current:. This function is documented on page 331.)

\s__color_stop Internal scan marks.

```
38631 \scan_new:N \s__color_stop
```

(End of definition for \s__color_stop.)

__color_select:N Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level material.

__color_select_math:N

__color_select:nn

```
38632 \cs_new_protected:Npn \__color_select:N #1
38633 {
38634   \exp_after:wN \__color_select:nn #1
38635   \group_insert_after:N \__color_backend_reset:
38636 }
38637 \cs_new_protected:Npn \__color_select_math:N #1
38638 { \exp_after:wN \__color_select:nn #1 }
38639 \cs_new_protected:Npn \__color_select:nn #1#2
38640 { \use:c { __color_backend_select_ #1 :n } {#2} }
```

(End of definition for __color_select:N, __color_select_math:N, and __color_select:nn.)

\l__color_current_tl The current color, with the model and

```
38641 \tl_new:N \l__color_current_tl
38642 \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
```

(End of definition for \l__color_current_tl.)

99.2 Predefined color names

The ability to predefine colors with a name is a key part of this module and means there has to be a method for storing the results. At first sight, it seems natural to follow the usual `expl3` model and create a `color` variable type for the process. That would then allow both local and global colors, constant colors and the like. However, these names need to be accessible in some form at the user level, for selection of colors either simply by name or as part of a more complex expression. This does not require that the full name is exposed but does require that they can be looked up in a predictable way. As such, it is more useful to expose just the color names as part of the interface, with the result that only local color names can be created. (This is also seen for example in key creation in `l3keys`.) As a result, color names are declarative (no `new` functions).

Since there is no need to manipulate colors *en masse*, each is stored in a two-part structure: a `prop` for the colors themselves, and a `tl` for the default model for each color.

99.3 Setup

```

\l__color_tmp_int
\l__color_tmp_tl
38643 \int_new:N \l__color_tmp_int
38644 \tl_new:N \l__color_tmp_tl

(End of definition for \l__color_tmp_int and \l__color_tmp_tl.)

\s__color_mark Internal scan marks. \s__color_stop is already defined in l3color-base.
38645 \scan_new:N \s__color_mark

(End of definition for \s__color_mark.)

\l__color_ignore_error_bool Used to avoid issuing multiple errors if there is a change-of-model with input container
an error.
38646 \bool_new:N \l__color_ignore_error_bool

(End of definition for \l__color_ignore_error_bool.)

```

99.4 Utility functions

\color_if_exist_p:n A simple wrapper to avoid needing to have the lookup repeated in too many places. To guard against a color created in a group, we need to test for entries in the `prop`.

\color_if_exist:nTF

```

38647 \prg_new_conditional:Npnn \color_if_exist:n #1 { p , T, F, TF }
38648 {
38649   \prop_if_exist:cTF { l__color_named_ #1 _prop }
38650   {
38651     \prop_if_empty:cTF { l__color_named_ #1 _prop }
38652     \prg_return_false:
38653     \prg_return_true:
38654   }
38655   \prg_return_false:
38656 }

(End of definition for \color_if_exist:nTF. This function is documented on page 334.)

```

```

\__color_model:N Simple abstractions.
\__color_values:N
38657 \cs_new:Npn \__color_model:N #1 { \exp_after:wN \use_i:nn #1 }
38658 \cs_new:Npn \__color_values:N #1 { \exp_after:wN \use_ii:nn #1 }

(End of definition for \__color_model:N and \__color_values:N.)

```

```

\__color_extract:nNN Recover the values for the standard model for a color.
\__color_extract:VNN
38659 \cs_new_protected:Npn \__color_extract:nNN #1#2#3
38660 {
38661   \tl_set_eq:Nc #2 { l__color_named_ #1 _tl }
38662   \prop_get:cVN { l__color_named_ #1 _prop } #2 #3
38663 }
38664 \cs_generate_variant:Nn \__color_extract:nNN { V }

(End of definition for \__color_extract:nNN.)

```

99.5 Model conversion

Model conversion is carried out using standard formulae for base models, as described in the manual for xcolor (see also the *PostScript Language Reference Manual*). For other models direct conversion might not be defined, so we go through the fallback models if necessary.

```

__color_convert:nnN
__color_convert:VVN
__color_convert:nnnN
__color_convert:nVnN
__color_convert:nnVN
38665 \cs_new_protected:Npn \__color_convert:nnN #1#2#3
38666 { \__color_convert:nnVN {#1} {#2} #3 #3 }
38667 \cs_generate_variant:Nn \__color_convert:nnN { VV }
38668 \cs_generate_variant:Nn \exp_last_unbraced:Nf { c }
38669 \cs_new_protected:Npn \__color_convert:nnnN #1#2#3#4
38670 {
38671   \tl_set:Nc #4
38672   {
38673     \cs_if_exist_use:CTF { __color_convert_ #1 _ #2 :w }
38674     { #3 \s__color_stop }
38675     {
38676       \cs_if_exist:CTF { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :
38677       {
38678         \exp_last_unbraced:cf
38679         { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :w }
38680         { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _tl } :w }
38681         \s__color_stop
38682       }
38683     }
38684     \exp_last_unbraced:cf
38685     { __color_convert_ \use:c { c__color_fallback_ #2 _tl } _ #2 :w }
38686     {
38687       \cs_if_exist_use:CTF { __color_convert_ #1 _ \use:c { c__color_fallback_
38688       { #3 \s__color_stop }
38689       {
38690         \exp_last_unbraced:cf
38691         { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ \use:c
38692         { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _
38693         \s__color_stop
38694       }
38695     }
38696     \s__color_stop
38697   }
38698 }
38699 }
38700 }
38701 \cs_generate_variant:Nn \__color_convert:nnnN { nV , nnV }
38702 \cs_new:Npn \__color_convert_gray_gray:w #1 \s__color_stop
38703 { #1 }
38704 \cs_new:Npn \__color_convert_gray_rgb:w #1 \s__color_stop
38705 { #1 ~ #1 ~ #1 }
38706 \cs_new:Npn \__color_convert_gray_cmyk:w #1 \s__color_stop
38707 { 0 ~ 0 ~ 0 ~ \fp_eval:n { 1 - #1 } }

These rather odd values are based on NTSC television: the set are used for the cmyk
conversion.
38708 \cs_new:Npn \__color_convert_rgb_gray:w #1 ~ #2 ~ #3 \s__color_stop
38709 { \fp_eval:n { 0.3 * #1 + 0.59 * #2 + 0.11 * #3 } }

```

```

38710 \cs_new:Npn \__color_convert_rgb_rgb:w #1 \s__color_stop
38711 { #1 }

```

The conversion from `rgb` to `cmk` is the most complex: a two-step procedure which requires *black generation* and *undercolor removal* functions. The PostScript reference describes them as device-dependent, but following `xcolor` we assume they are linear. Moreover, as the likelihood of anyone using a non-unitary matrix here is tiny, we simplify and treat those two concepts as no-ops. To allow code sharing with parsing of `cmk` values, we have an intermediate function here (`\__color_convert_rgb_cmyk:nnn`) which actually takes `cmk` values as input.

```

38712 \cs_new:Npn \__color_convert_rgb_cmyk:w #1 ~ #2 ~ #3 \s__color_stop
38713 {
38714   \exp_args:Neee \__color_convert_rgb_cmyk:nnn
38715   { \fp_eval:n { 1 - #1 } }
38716   { \fp_eval:n { 1 - #2 } }
38717   { \fp_eval:n { 1 - #3 } }
38718 }
38719 \cs_new:Npn \__color_convert_rgb_cmyk:nnn #1#2#3
38720 {
38721   \exp_args:Ne \__color_convert_rgb_cmyk:nnnn
38722   { \fp_eval:n { min ( #1 , #2 , #3 ) } } {#1} {#2} {#3}
38723 }
38724 \cs_new:Npn \__color_convert_rgb_cmyk:nnnn #1#2#3#4
38725 {
38726   \fp_eval:n { min ( 1 , max ( 0 , #2 - #1 ) ) } \c_space_tl
38727   \fp_eval:n { min ( 1 , max ( 0 , #3 - #1 ) ) } \c_space_tl
38728   \fp_eval:n { min ( 1 , max ( 0 , #4 - #1 ) ) } \c_space_tl
38729   #1
38730 }
38731 \cs_new:Npn \__color_convert_cmyk_gray:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
38732 { \fp_eval:n { 1 - min ( 1 , 0.3 * #1 + 0.59 * #2 + 0.11 * #3 + #4 ) } }
38733 \cs_new:Npn \__color_convert_cmyk_rgb:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
38734 {
38735   \fp_eval:n { 1 - min ( 1 , #1 + #4 ) } \c_space_tl
38736   \fp_eval:n { 1 - min ( 1 , #2 + #4 ) } \c_space_tl
38737   \fp_eval:n { 1 - min ( 1 , #3 + #4 ) }
38738 }
38739 \cs_new:Npn \__color_convert_cmyk_cmyk:w #1 \s__color_stop
38740 { #1 }

```

(End of definition for `\__color_convert:nnN` and others.)

99.6 Color expressions

<pre> \l__color_model_tl \l__color_value_tl \l__color_next_model_tl \l__color_next_value_tl </pre>	Working space to store the color data whilst doing calculations: keeping it on the stack is attractive but gets tricky (return is non-trivial). <pre> 38741 \tl_new:N \l__color_model_tl 38742 \tl_new:N \l__color_value_tl 38743 \tl_new:N \l__color_next_model_tl 38744 \tl_new:N \l__color_next_value_tl </pre>

(End of definition for `\l__color_model_tl` and others.)

```

    \_color_parse:nN
    \_color_parse_aux:nN
    \_color_parse_eq:Nn
    \_color_parse_eq:nNn
    \_color_parse:Nw
    \_color_parse_loop_init:Nnn
    \_color_parse_loop:w
    \_color_parse_loop_check:nn
    \_color_parse_loop:nn
    \_color_parse_gray:n
    \_color_parse_std:n
    \_color_parse_break:w
    \_color_parse_end:
    \_color_parse_mix:Nnnn
    \_color_parse_mix:NVVn
    \_color_parse_mix:nNnn
    \_color_parse_mix_gray:nw
    \_color_parse_mix_rgb:nw
    \_color_parse_mix_cmyk:nw

```

The main function for parsing color expressions removes actives but otherwise expands, then starts working through the expression itself. At the end, we apply the payload.

```

38745 \cs_new_protected:Npe \_color_parse:nN #1#2
38746 {
38747   \tl_set:Nc \exp_not:c { l__color_named_ . _tl }
38748   { \exp_not:N \_color_model:N \exp_not:N \l__color_current_tl }
38749   \prop_put:Nve \exp_not:c { l__color_named_ . _prop }
38750   \exp_not:c { l__color_named_ . _tl }
38751   { \exp_not:N \_color_values:N \exp_not:N \l__color_current_tl }
38752   \exp_not:N \exp_args:Nc \exp_not:N \_color_parse_aux:nN
38753   { \exp_not:N \tl_to_str:n {#1} } #2
38754 }

```

Before going to all of the effort of parsing an expression, these two precursor functions look for a pre-defined name, either on its own or with a trailing ! (which is the same thing).

```

38755 \cs_new_protected:Npn \_color_parse_aux:nN #1#2
38756 {
38757   \color_if_exist:nTF {#1}
38758   { \_color_parse_set_eq:Nn #2 {#1} }
38759   { \_color_parse:Nw #2#1 ! \s__color_stop }
38760   \_color_check_model:N #2
38761 }
38762 \cs_new_protected:Npn \_color_parse_set_eq:Nn #1#2
38763 {
38764   \tl_if_empty:NTF \l_color_fixed_model_tl
38765   { \exp_args:Nv \_color_parse_set_eq:nNn { l__color_named_ #2 _tl } }
38766   { \exp_args:Nv \_color_parse_set_eq:nNn \l_color_fixed_model_tl }
38767   #1 {#2}
38768 }

```

Here, we have to allow for the case where there is a fixed model: that can't be swept up by generic conversion as we are dealing with a named color.

```

38769 \cs_new_protected:Npn \_color_parse_set_eq:nNn #1#2#3
38770 {
38771   \prop_get:cnNTF
38772   { l__color_named_ #3 _prop } {#1}
38773   \l__color_value_tl
38774   { \tl_set:Nc #2 { {#1} { \l__color_value_tl } } }
38775   {
38776     \tl_set_eq:Nc \l__color_model_tl { l__color_named_ #3 _tl }
38777     \prop_get:cVN { l__color_named_ #3 _prop } \l__color_model_tl
38778     \l__color_value_tl
38779     \_color_convert:nnN
38780     \l__color_model_tl {#1} \l__color_value_tl
38781     \tl_set:Nc #2
38782     {
38783       {#1}
38784       { \l__color_value_tl }
38785     }
38786   }
38787 }
38788 \cs_new_protected:Npn \_color_parse:Nw #1#2 ! #3 \s__color_stop
38789 {

```

```

38790 \color_if_exist:nTF {#2}
38791 {
38792   \tl_if_blank:nTF {#3}
38793   { \__color_parse_set_eq:Nn #1 {#2} }
38794   { \__color_parse_loop_init:Nnn #1 {#2} {#3} }
38795 }
38796 {
38797   \msg_error:nnn { color } { unknown-color } {#2}
38798   \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
38799 }
38800 }

```

Once we establish that a full parse is needed, the next job is to get the detail of the first color. That will determine the model we use for the calculation: splitting here makes checking that a bit easier.

```

38801 \cs_new_protected:Npn \__color_parse_loop_init:Nnn #1#2#3
38802 {
38803   \group_begin:
38804     \__color_extract:nNN {#2} \l__color_model_tl \l__color_value_tl
38805     \__color_parse_loop:w #3 ! ! ! \s__color_stop
38806     \tl_set:Nc \l__color_tmp_tl
38807     { { \l__color_model_tl } { \l__color_value_tl } }
38808     \exp_args:NNNV \group_end:
38809     \tl_set:Nn #1 \l__color_tmp_tl
38810 }

```

This is the loop proper: there can be an open-ended set of colors to parse, separated by ! tokens. There are a few cases to look out for. At the end of the expression and with we find a mix of 100 then we simply skip the next color entirely (we can't stop the loop as there might be a further valid color to mix in). On the other hand, if we get a mix of 0 then drop everything so far and start again. There is also a trailing **white** to “read in” if the final explicit data is a mix. Those conditions are separate from actually looping, which is therefore sorted out by checking if we have further data to process: in contrast to **xcolor**, we don't allow !! so the test can be simplified.

```

38811 \cs_new_protected:Npn \__color_parse_loop:w #1 ! #2 ! #3 ! #4 ! #5 \s__color_stop
38812 {
38813   \tl_if_blank:nF {#1}
38814   {
38815     \bool_lazy_and:nnTF
38816     { \fp_compare_p:nNn {#1} > { 0 } }
38817     { \fp_compare_p:nNn {#1} < { 100 } }
38818     {
38819       \use:e
38820       {
38821         \__color_parse_loop:nn {#1}
38822         { \tl_if_blank:nTF {#2} { white } {#2} }
38823       }
38824     }
38825     { \__color_parse_loop_check:nn {#1} {#2} }
38826   }
38827   \tl_if_blank:nF {#3}
38828   { \__color_parse_loop:w #3 ! #4 ! #5 \s__color_stop }
38829   \__color_parse_end:
38830 }

```

As these are unusual cases, we accept slower performance here for clearer code: check for the error conditions, handle the boundary cases after that.

```

38831 \cs_new_protected:Npn \__color_parse_loop_check:nn #1#2
38832 {
38833   \bool_if:NF \l__color_ignore_error_bool
38834   {
38835     \bool_lazy_or:nnT
38836     { \fp_compare_p:nNn {#1} < { 0 } }
38837     { \fp_compare_p:nNn {#1} > { 100 } }
38838     { \msg_error:nnnnn { color } { out-of-range } {#1} { 0 } { 100 } }
38839   }
38840   \fp_compare:nNnF {#1} > \c_zero_fp
38841   {
38842     \tl_if_blank:nTF {#2}
38843     { \__color_extract:nnN { white } }
38844     { \__color_extract:nnN {#2} }
38845     \l__color_model_tl \l__color_value_tl
38846   }
38847 }

```

The “payload” of calculation in the loop first. If the model for the upcoming color is different from that of the existing (partial) color, convert the model. For **gray** the two are flipped round so that the outcome is something with “real” color. We are then in a position to do the actual calculation itself. The two auxiliaries here give us a way to break the loop should an invalid name be found.

```

38848 \cs_new_protected:Npn \__color_parse_loop:nn #1#2
38849 {
38850   \color_if_exist:nTF {#2}
38851   {
38852     \__color_extract:nnN {#2} \l__color_next_model_tl \l__color_next_value_tl
38853     \tl_if_eq:NNF \l__color_model_tl \l__color_next_model_tl
38854     {
38855       \str_if_eq:VnTF \l__color_model_tl { gray }
38856       { \__color_parse_gray:n {#2} }
38857       { \__color_parse_std:n {#2} }
38858     }
38859     \tl_set:Ne \l__color_value_tl
38860     {
38861       \__color_parse_mix:NVVn
38862       \l__color_model_tl \l__color_value_tl \l__color_next_value_tl {#1}
38863     }
38864   }
38865   {
38866     \msg_error:nnn { color } { unknown-color } {#2}
38867     \__color_extract:nnN { black } \l__color_model_tl \l__color_value_tl
38868     \__color_parse_break:w
38869   }
38870 }

```

The **gray** model needs special handling: the models need to be swapped: we do that using a dedicated function.

```

38871 \cs_new_protected:Npn \__color_parse_gray:n #1
38872 {
38873   \tl_set_eq:NN \l__color_model_tl \l__color_next_model_tl

```



```

38874     \tl_set:Nn \l__color_next_model_tl { gray }
38875     \exp_args:NnV \__color_convert:nnN { gray } \l__color_model_tl
38876     \l__color_value_tl
38877     \prop_get:cVN { l__color_named_ #1 _prop } \l__color_model_tl
38878     \l__color_next_value_tl
38879   }
38880 \cs_new_protected:Npn \__color_parse_std:n #1
38881 {
38882   \prop_get:cVNF { l__color_named_ #1 _prop }
38883   \l__color_model_tl
38884   \l__color_next_value_tl
38885   {
38886     \__color_convert:VNN
38887     \l__color_next_model_tl
38888     \l__color_model_tl
38889     \l__color_next_value_tl
38890   }
38891 }
38892 \cs_new_protected:Npn \__color_parse_break:w #1 \__color_parse_end: { }
38893 \cs_new_protected:Npn \__color_parse_end: { }

```

Do the vector arithmetic: mainly a question of shuffling input, along with one pre-calculation to keep down the use of division.

```

38894 \cs_new:Npn \__color_parse_mix:Nnnn #1#2#3#4
38895 {
38896   \exp_args:Nf \__color_parse_mix:nNnn
38897   { \fp_eval:n { #4 / 100 } }
38898   #1 {#2} {#3}
38899 }
38900 \cs_generate_variant:Nn \__color_parse_mix:Nnnn { NVV }
38901 \cs_new:Npn \__color_parse_mix:nNnn #1#2#3#4
38902 {
38903   \use:c { __color_parse_mix_ #2 :nw } {#1}
38904   #3 \s__color_mark #4 \s__color_stop
38905 }
38906 \cs_new:Npn \__color_parse_mix_gray:nw #1#2 \s__color_mark #3 \s__color_stop
38907 { \fp_eval:n { #2 * #1 + #3 * ( 1 - #1 ) } }
38908 \cs_new:Npn \__color_parse_mix_rgb:nw
38909 #1#2 ~ #3 ~ #4 \s__color_mark #5 ~ #6 ~ #7 \s__color_stop
38910 {
38911   \fp_eval:n { #2 * #1 + #5 * ( 1 - #1 ) } \c_space_tl
38912   \fp_eval:n { #3 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
38913   \fp_eval:n { #4 * #1 + #7 * ( 1 - #1 ) }
38914 }
38915 \cs_new:Npn \__color_parse_mix_cmyk:nw
38916 #1#2 ~ #3 ~ #4 ~ #5 \s__color_mark #6 ~ #7 ~ #8 ~ #9 \s__color_stop
38917 {
38918   \fp_eval:n { #2 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
38919   \fp_eval:n { #3 * #1 + #7 * ( 1 - #1 ) } \c_space_tl
38920   \fp_eval:n { #4 * #1 + #8 * ( 1 - #1 ) } \c_space_tl
38921   \fp_eval:n { #5 * #1 + #9 * ( 1 - #1 ) }
38922 }

```

(End of definition for __color_parse:nN and others.)

```

\__color_parse_model_gray:w Turn the input into internal form, also tidying up the number quickly.
\__color_parse_model_rgb:w 38923 \cs_new:Npn \__color_parse_model_gray:w #1 , #2 \s__color_stop
\__color_parse_model_cmyk:w 38924 { { gray } { \__color_parse_number:n {#1} } }
\__color_parse_number:n 38925 \cs_new:Npn \__color_parse_model_rgb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_number:w 38926 {
38927 { rgb }
38928 {
38929 \__color_parse_number:n {#1} ~
38930 \__color_parse_number:n {#2} ~
38931 \__color_parse_number:n {#3}
38932 }
38933 }
38934 \cs_new:Npn \__color_parse_model_cmyk:w #1 , #2 , #3 , #4 , #5 \s__color_stop
38935 {
38936 { cmyk }
38937 {
38938 \__color_parse_number:n {#1} ~
38939 \__color_parse_number:n {#2} ~
38940 \__color_parse_number:n {#3} ~
38941 \__color_parse_number:n {#4}
38942 }
38943 }
38944 \cs_new:Npn \__color_parse_number:n #1
38945 { \__color_parse_number:w #1 . 0 . \s__color_stop }
38946 \cs_new:Npn \__color_parse_number:w #1 . #2 . #3 \s__color_stop
38947 { \tl_if_blank:nTF {#1} { 0 } {#1} . #2 }

```

(End of definition for __color_parse_model_gray:w and others.)

```

\__color_parse_model_Gray:w 38948 \cs_new:Npn \__color_parse_model_Gray:w #1 , #2 \s__color_stop
\__color_parse_model_hsb:w 38949 { { gray } { \fp_eval:n { #1 / 15 } } }
\__color_parse_model_Hsb:w 38950 \cs_new:Npn \__color_parse_model_hsb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_model_HSB:w 38951 { \__color_parse_model_hsb:nnn {#1} {#2} {#3} }
\__color_parse_model_HTML:w 38952 \cs_new:Npn \__color_parse_model_Hsb:w #1 , #2 , #3 , #4 \s__color_stop
\__color_parse_model_RGB:w 38953 {
\__color_parse_model_hsb:nnn 38954 \exp_args:Ne \__color_parse_model_hsb:nnn { \fp_eval:n { #1 / 360 } }
\__color_parse_model_hsb:nnnn 38955 {#2} {#3}
\__color_parse_model_hsb:nnnnn 38956 }
\__color_parse_model_hsb_0:nnnn The conversion here is non-trivial but is described at length in the xcolor manual. For
\__color_parse_model_hsb_1:nnnn ease, we calculate the integer and fractional parts of the hue first, then use them to work
\__color_parse_model_hsb_2:nnnn out the possible values for r, g and b before putting them in the correct places.
\__color_parse_model_hsb_3:nnnn 38957 \cs_new:Npn \__color_parse_model_hsb:nnn #1#2#3
\__color_parse_model_hsb_4:nnnn 38958 {
\__color_parse_model_hsb_5:nnnn 38959 { rgb }
38960 {
38961 \exp_args:Ne \__color_parse_model_hsb_aux:nnn
38962 { \fp_eval:n { 6 * (#1) } } {#2} {#3}
38963 }
38964 }
38965 \cs_new:Npn \__color_parse_model_hsb_aux:nnn #1#2#3
38966 {
38967 \exp_args:Nee \__color_parse_model_hsb_aux:nnnn

```

```

38968     { \fp_eval:n { floor(#1) } } { \fp_eval:n { #1 - floor(#1) } }
38969     {#2} {#3}
38970 }
38971 \cs_new:Npn \__color_parse_model_hsb_aux:nnnn #1#2#3#4
38972 {
38973     \use:e
38974     {
38975         \exp_not:N \__color_parse_model_hsb_aux:nnnnn
38976         { \__color_parse_number:n {#4} }
38977         { \fp_eval:n { round(#4 * (1 - #3), 5) } }
38978         { \fp_eval:n { round(#4 * (1 - #3 * #2), 5) } }
38979         { \fp_eval:n { round(#4 * (1 - #3 * (1 - #2)), 5) } }
38980         {#1}
38981     }
38982 }
38983 \cs_new:Npn \__color_parse_model_hsb_aux:nnnnn #1#2#3#4#5
38984 { \use:c { __color_parse_model_hsb_ #5 :nnnn } {#1} {#2} {#3} {#4} }
38985 \cs_new:cpn { __color_parse_model_hsb_0:nnnn } #1#2#3#4 { #1 ~ #4 ~ #2 }
38986 \cs_new:cpn { __color_parse_model_hsb_1:nnnn } #1#2#3#4 { #3 ~ #1 ~ #2 }
38987 \cs_new:cpn { __color_parse_model_hsb_2:nnnn } #1#2#3#4 { #2 ~ #1 ~ #4 }
38988 \cs_new:cpn { __color_parse_model_hsb_3:nnnn } #1#2#3#4 { #2 ~ #3 ~ #1 }
38989 \cs_new:cpn { __color_parse_model_hsb_4:nnnn } #1#2#3#4 { #4 ~ #2 ~ #1 }
38990 \cs_new:cpn { __color_parse_model_hsb_5:nnnn } #1#2#3#4 { #1 ~ #2 ~ #3 }
38991 \cs_new:cpn { __color_parse_model_hsb_6:nnnn } #1#2#3#4 { #1 ~ #2 ~ #2 }
38992 \cs_new:Npn \__color_parse_model_HSB:w #1 , #2 , #3 , #4 \s__color_stop
38993 {
38994     \exp_args:Neee \__color_parse_model_hsb:nnn
38995     { \fp_eval:n { round((#1) / 240, 5) } }
38996     { \fp_eval:n { round((#2) / 240, 5) } }
38997     { \fp_eval:n { round((#3) / 240, 5) } }
38998 }
38999 \cs_new:Npn \__color_parse_model_HTML:w #1 , #2 \s__color_stop
39000 { \__color_parse_model_HTML_aux:w #1 0 0 0 0 0 0 \s__color_stop }
39001 \cs_new:Npn \__color_parse_model_HTML_aux:w #1#2#3#4#5#6#7 \s__color_stop
39002 {
39003     { rgb }
39004     {
39005         \fp_eval:n { round(\int_from_hex:n {#1#2} / 255, 5) } ~
39006         \fp_eval:n { round(\int_from_hex:n {#3#4} / 255, 5) } ~
39007         \fp_eval:n { round(\int_from_hex:n {#5#6} / 255, 5) }
39008     }
39009 }
39010 \cs_new:Npn \__color_parse_model_RGB:w #1 , #2 , #3 , #4 \s__color_stop
39011 {
39012     { rgb }
39013     {
39014         \fp_eval:n { round((#1) / 255, 5) } ~
39015         \fp_eval:n { round((#2) / 255, 5) } ~
39016         \fp_eval:n { round((#3) / 255, 5) }
39017     }
39018 }

```

Following the description in the xcolor manual. As we always use `rgb`, there is no need to find the sixth, we just pass the information straight to the `hsb` auxiliary defined earlier.

```

39019 \cs_new:Npn \__color_parse_model_wave:w #1 , #2 \s__color_stop
39020 {
39021   { rgb }
39022   {
39023     \fp_compare:nNnTF {#1} < { 420 }
39024     { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 380) / 40 }
39025       }
39026     {
39027       \fp_compare:nNnTF {#1} > { 700 }
39028       { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 780) / -80 } }
39029       { \__color_parse_model_wave_auxi:nn {#1} { 1 } }
39030     }
39031   }
39032 }
39033 \cs_new:Npn \__color_parse_model_wave_auxi:nn #1#2
39034 {
39035   \fp_compare:nNnTF {#1} < { 440 }
39036   {
39037     \__color_parse_model_wave_auxii:nn
39038     { 4 + \__color_parse_model_wave_rho:n { (#1 - 440) / -60 } }
39039     {#2}
39040   }
39041   {
39042     \fp_compare:nNnTF {#1} < { 490 }
39043     {
39044       \__color_parse_model_wave_auxii:nn
39045       { 4 - \__color_parse_model_wave_rho:n { (#1 - 440) / 50 } }
39046       {#2}
39047     }
39048     {
39049       \fp_compare:nNnTF {#1} < { 510 }
39050       {
39051         \__color_parse_model_wave_auxii:nn
39052         { 2 + \__color_parse_model_wave_rho:n { (#1 - 510) / -20 } }
39053         {#2}
39054       }
39055       {
39056         \fp_compare:nNnTF {#1} < { 580 }
39057         {
39058           \__color_parse_model_wave_auxii:nn
39059           { 2 - \__color_parse_model_wave_rho:n { (#1 - 510) / 70 } }
39060           {#2}
39061         }
39062         {
39063           \fp_compare:nNnTF {#1} < { 645 }
39064           {
39065             \__color_parse_model_wave_auxii:nn
39066             { \__color_parse_model_wave_rho:n { (#1 - 645) / -65 } }
39067             {#2}
39068           }
39069           { \__color_parse_model_wave_auxii:nn { 0 } {#2} }
39070         }
39071       }
39072     }

```

```

39073     }
39074   }
39075 \cs_new:Npn \__color_parse_model_wave_auxii:nn #1#2
39076 {
39077   \exp_args:Neee \__color_parse_model_hsb_aux:nnn
39078     { \fp_eval:n {#1} }
39079     { 1 }
39080     { \__color_parse_model_wave_rho:n {#2} }
39081 }
39082 \cs_new:Npn \__color_parse_model_wave_rho:n #1
39083 { \fp_eval:n { min(1, max(0,#1) ) } }

```

(End of definition for __color_parse_model_Gray:w and others.)

__color_parse_model_cmy:w Simply pass data to the conversion functions.

```

39084 \cs_new:Npn \__color_parse_model_cmy:w #1 , #2 , #3 , #4 \s__color_stop
39085 {
39086   { cmyk }
39087   { \__color_convert_rgb_cmyk:nnn {#1} {#2} {#3} }
39088 }

```

(End of definition for __color_parse_model_cmy:w.)

__color_parse_model_tHsb:w There are three stages to the process here: bring the tH argument into the normal range, divide through to get to hsb and finally convert that to rgb. The final stage can be delegated to the parsing function for hsb, and the conversion from Hsb to hsb is trivial, so the main focus here is the first stage. We use a simple expandable loop to do the work, and we implement the equation given in the xcolor manual (number 85 there) as a simple expression.

```

39089 \cs_new:Npn \__color_parse_model_tHsb:w #1 , #2 , #3 , #4 \s__color_stop
39090 {
39091   \exp_args:Ne \__color_parse_model_hsb:nnn
39092     { \__color_parse_model_tHsb:n {#1} } {#2} {#3}
39093 }
39094 \cs_new:Npn \__color_parse_model_tHsb:n #1
39095 {
39096   \__color_parse_model_tHsb:nw {#1}
39097     0 , 0 ;
39098     60 , 30 ;
39099     120 , 60 ;
39100     180 , 120 ;
39101     210 , 180 ;
39102     240 , 240 ;
39103     360 , 360 ;
39104     \q_recursion_tail , ;
39105     \q_recursion_stop
39106 }
39107 \cs_new:Npn \__color_parse_model_tHsb:nw #1 #2 , #3 ; #4 , #5 ;
39108 {
39109   \quark_if_recursion_tail_stop_do:nn {#4} { 0 }
39110   \fp_compare:nNnTF {#1} > {#4}
39111     { \__color_parse_model_tHsb:nw {#1} #4 , #5 ; }
39112     {
39113       \use_i_delimit_by_q_recursion_stop:nw

```

```

39114         { \fp_eval:n { ((#1 - #2) / (#4 - #2) * (#5 - #3) + #3) / 360 } }
39115     }
39116 }

```

(End of definition for `\__color_parse_model_tHsb:w`, `\__color_parse_model_tHsb:n`, and `\__color_parse_model_tHsb:nw`.)

`\__color_parse_model_&spot:w` We cannot extract data here from that passed by `xcolor`, so we fall back on a black tint.

```

39117 \cs_new:cpn { __color_parse_model_&spot:w } #1 , #2 \s__color_stop
39118 { { gray } { #1 } }

```

(End of definition for `\__color_parse_model_&spot:w`.)

`\__color_parse_model_linearrgb:nnn`
`\__color_parse_model_linearrgb_aux:n`

Encode the tristimulus values of the sRGB primaries according to sRGB. That is, take the actual physical light intensities of the primaries defined by the default RGB color space sRGB ([IEC 61966-2-1](#)) and transform them with the gamma function defined in the same standard¹⁴:

$$x_{\text{standard}} = \begin{cases} 12.92 \cdot x & x < 0.0031308 \\ 1.055 \cdot x^{1/2.4} - 0.055 & x \geq 0.0031308 \end{cases}$$

Additionally clip values to the range $[0, 1]$.

```

39119 \cs_new:Npn \__color_parse_model_linearrgb:nnn #1#2#3
39120 {
39121     { rgb }
39122     {
39123         \__color_parse_model_linearrgb_aux:n { #1 } ~
39124         \__color_parse_model_linearrgb_aux:n { #2 } ~
39125         \__color_parse_model_linearrgb_aux:n { #3 }
39126     }
39127 }
39128 \cs_new:Npn \__color_parse_model_linearrgb_aux:n #1
39129 {
39130     \fp_compare:nNnTF { #1 } < { 0.0031308 }
39131     { \fp_eval:n { 12.92 * max(0, #1) } }
39132     { \fp_eval:n { 1.055 * min(1, #1) ^ 0.4166666666666667 - 0.055 } }
39133 }

```

(End of definition for `\__color_parse_model_linearrgb:nnn` and `\__color_parse_model_linearrgb_aux:n`.)

`\__color_parse_model_lms:nnn`
`\__color_parse_model_cbrtlms:nnn`
`\__color_parse_model_oklab:nnn`
`\__color_parse_model_oklab:w`

The [Oklab color space](#) is defined by first transforming the physical light intensity to the biophysical responses of the L, M and S cones in a typical human eye, then taking those responses to the power of $\frac{1}{3}$ (which approximately fits human lightness perception), and finally applying one more matrix transformation M_2 to obtain handy L , a , and b values. Here, we just go the other way round:

First, transform L , a and b into the cube roots of l , m and s cone response intensities using the inverse of Oklab's defining matrix M_2 :

$$\begin{pmatrix} l' \\ m' \\ s' \end{pmatrix} = \begin{pmatrix} 1 & 0.3963377774 & 0.2158037573 \\ 1 & -0.1055613458 & -0.0638541728 \\ 1 & -0.0894841775 & -1.2914855480 \end{pmatrix} \begin{pmatrix} L \\ a \\ b \end{pmatrix}$$

¹⁴For background information see <https://www.w3.org/Graphics/Color/sRGB>.

```

39134 \cs_new:Npn \__color_parse_model_oklab:nnn #1#2#3
39135 {
39136   \exp_args:Neee \__color_parse_model_cbrtlms:nnn
39137   { \fp_eval:n { #1 + 0.3963377774 * #2 + 0.2158037573 * #3 } }
39138   { \fp_eval:n { #1 - 0.1055613458 * #2 - 0.0638541728 * #3 } }
39139   { \fp_eval:n { #1 - 0.0894841775 * #2 - 1.2914855480 * #3 } }
39140 }

```

Next, take those cube roots to the power of 3 to obtain the actual cone responses:

$$l = l'^3, \quad m = m'^3, \quad s = s'^3$$

```

39141 \cs_new:Npn \__color_parse_model_cbrtlms:nnn #1#2#3
39142 {
39143   \exp_args:Neee \__color_parse_model_lms:nnn
39144   { \fp_eval:n { #1 ^ 3 } }
39145   { \fp_eval:n { #2 ^ 3 } }
39146   { \fp_eval:n { #3 ^ 3 } }
39147 }

```

Finally, transform l , m and s cone responses to linear sRGB intensity.

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \begin{pmatrix} 4.0767416621 & -3.3077115913 & 0.2309699292 \\ -1.2684380046 & 2.6097574011 & -0.3413193965 \\ -0.0041960863 & -0.7034186147 & 1.7076147010 \end{pmatrix} \begin{pmatrix} l \\ m \\ s \end{pmatrix}$$

```

39148 \cs_new:Npn \__color_parse_model_lms:nnn #1#2#3
39149 {
39150   \exp_args:Neee \__color_parse_model_linearrgb:nnn
39151   { \fp_eval:n { 4.0767416621 * #1 - 3.3077115913 * #2 + 0.2309699292 * #3 } }
39152   { \fp_eval:n { -1.2684380046 * #1 + 2.6097574011 * #2 - 0.3413193965 * #3 } }
39153   { \fp_eval:n { -0.0041960863 * #1 - 0.7034186147 * #2 + 1.7076147010 * #3 } }
39154 }

```

The actual parse macro with *weird* signature.

```

39155 \cs_new:Npn \__color_parse_model_oklab:w #1 , #2 , #3 , #4 \s__color_stop
39156 { \__color_parse_model_oklab:nnn { #1 } { #2 } { #3 } }

```

Oklch is Oklab in cylinder coordinates with lightness $L \in [0, 1]$, chroma c giving the color intensity in values up to around 0.35 and hue h given in degrees (typically given as value from 0 to 360):

$$L = L, \quad a = h \cdot \cos(c), \quad b = h \cdot \sin(c)$$

```

39157 \cs_new:Npn \__color_parse_model_oklch:w #1 , #2 , #3 , #4 \s__color_stop
39158 {
39159   \exp_args:Nnee \__color_parse_model_oklab:nnn
39160   {#1}
39161   { \fp_eval:n { #2 * cosd(#3) } }
39162   { \fp_eval:n { #2 * sind(#3) } }
39163 }

```

(End of definition for `\__color_parse_model_lms:nnn` and others.)

99.7 Selecting colors (and color models)

`\l_color_fixed_model_tl` For selecting a single fixed model.

```
39164 \tl_new:N \l_color_fixed_model_tl
```

(End of definition for `\l_color_fixed_model_tl`. This variable is documented on page 335.)

`\__color_check_model:N` Check that the model in use is the one required.

```
\__color_check_model:nn
39165 \cs_new_protected:Npn \__color_check_model:N #1
39166 {
39167   \tl_if_empty:NF \l_color_fixed_model_tl
39168   {
39169     \exp_after:wN \__color_check_model:nn #1
39170     \tl_if_eq:NNF \l__color_model_tl \l_color_fixed_model_tl
39171     {
39172       \__color_convert:VWN \l__color_model_tl \l_color_fixed_model_tl
39173       \l__color_value_tl
39174     }
39175     \tl_set:Nc #1
39176     { { \l_color_fixed_model_tl } { \l__color_value_tl } }
39177   }
39178 }
39179 \cs_new_protected:Npn \__color_check_model:nn #1#2
39180 {
39181   \tl_set:Nn \l__color_model_tl {#1}
39182   \tl_set:Nn \l__color_value_tl {#2}
39183 }
```

(End of definition for `\__color_check_model:N` and `\__color_check_model:nn`.)

`\__color_finalize_current:` A backend-neutral location for “last minute” manipulations before handing off to the backend code. We set the special . syntax here: this will therefore always be available. The finalization is separate from the main function so it can also be applied to *e.g.* page color.

```
39184 \cs_new_protected:Npe \__color_finalize_current:
39185 {
39186   \tl_set:Nc \exp_not:c { l__color_named_ . _tl }
39187   { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
39188   \prop_clear:N \exp_not:c { l__color_named_ . _prop }
39189   \prop_put:Nve \exp_not:c { l__color_named_ . _prop }
39190   \exp_not:c { l__color_named_ . _tl }
39191   { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
39192 }
```

(End of definition for `\__color_finalize_current:`.)

`\color_select:n` Parse the input expressions then get the backend to actually activate them. The main complexity here is the need to check through multiple models. That is done “locally” here as the approach is subtly different to when different models are being stored.

```
\color_select:V
\color_select:nn
\color_select:nV
\color_select:Vn
\color_select:VV
\__color_select_main:Nnn
\__color_select_main:Nw
\__color_select_loop:Nw
\__color_select:nnN
\__color_select_swap:Nnn
39193 \cs_new_protected:Npn \color_select:n #1
39194 {
39195   \__color_parse:nN {#1} \l__color_current_tl
39196   \__color_finalize_current:
39197   \__color_select:N \l__color_current_tl
```



```

39198 }
39199 \cs_generate_variant:Nn \color_select:n { V }
39200 \cs_new_protected:Npn \color_select:nn #1#2
39201 {
39202   \__color_select_main:Nnn \l__color_current_tl {#1} {#2}
39203   \__color_finalize_current:
39204   \__color_select:N \l__color_current_tl
39205 }
39206 \cs_generate_variant:Nn \color_select:nn { nV , V , VV }

```

If the first color model is the fixed one, or if there is no fixed model, we don't need most of the data: just set up and apply the backend function.

```

39207 \cs_new_protected:Npn \__color_select_main:Nnn #1#2#3
39208 {
39209   \use:e
39210   {
39211     \exp_not:N \__color_select_main:Nw \exp_not:N #1
39212     \exp_not:n {#2} / / \exp_not:N \s__color_mark
39213     #3 / / \exp_not:N \s__color_stop
39214   }
39215 }
39216 \cs_new_protected:Npn \__color_select_main:Nw
39217 #1 #2 / #3 / #4 \s__color_mark #5 / #6 / #7 \s__color_stop
39218 {
39219   \__color_select:nnN {#2} {#5} #1
39220   \bool_lazy_or:nnF
39221   { \tl_if_empty_p:N \l_color_fixed_model_tl }
39222   { \str_if_eq_p:nV {#2} \l_color_fixed_model_tl }
39223   { \__color_select_loop:Nw #1 #3 / #4 \s__color_mark #6 / #7 \s__color_stop }
39224 }

```

If a fixed model applies, we need to check each possible value in order. If there is no hit at all, fall back on the generic formula-based interchange.

```

39225 \cs_new_protected:Npn \__color_select_loop:Nw
39226 #1 #2 / #3 \s__color_mark #4 / #5 \s__color_stop
39227 {
39228   \str_if_eq:nVTF {#2} \l_color_fixed_model_tl
39229   { \__color_select:nnN {#2} {#4} #1 }
39230   {
39231     \tl_if_blank:nTF {#2}
39232     { \exp_after:wN \__color_select_swap:Nnn \exp_after:wN #1 #1 }
39233     { \__color_select_loop:Nw #1 #3 \s__color_mark #5 \s__color_stop }
39234   }
39235 }
39236 \cs_new_protected:Npn \__color_select:nnN #1#2#3
39237 {
39238   \cs_if_exist:cTF { __color_parse_model_ #1 :w }
39239   {
39240     \tl_set:Ne #3
39241     { \use:c { __color_parse_model_ #1 :w } #2 , 0 , 0 , 0 , 0 \s__color_stop }
39242   }
39243   { \msg_error:nnn { color } { unknown-model } {#1} }
39244 }
39245 \cs_new_protected:Npn \__color_select_swap:Nnn #1#2#3
39246 {

```

```

39247   \_color_convert:nVnN {#2} \l_color_fixed_model_tl {#3} \l__color_value_tl
39248   \tl_set:Ne #1
39249       { { \l_color_fixed_model_tl } { \l__color_value_tl } }
39250   }

```

(End of definition for `\color_select:n` and others. These functions are documented on page 335.)

99.8 Math color

The approach here is the same as for the L^AT_EX_{2 ϵ} `\mathcolor` command, but as we are working at the expl3 level we can make some minor changes.

`\l_color_math_active_tl` Tokens representing active sub/superscripts.

```

39251 \tl_new:N \l_color_math_active_tl
39252 \tl_set:Nn \l_color_math_active_tl { ' }

```

(End of definition for `\l_color_math_active_tl`. This function is documented on page 336.)

`\g__color_math_seq` Not all engines have multiple color stacks, and at the same time we are not expecting breaking within a colored math fragment. So we track the color stack ourselves.

```

39253 \seq_new:N \g__color_math_seq

```

(End of definition for `\g__color_math_seq`.)

`\color_math:nn` The basic set up here is relatively simple: store the current color, parse the new color
`\color_math:nnn` as-normal, then switch color before inserting the tokens we are asked to change. The
`\__color_math:nn` tricky part is right at the end, handling the reset.

```

39254 \cs_new_protected:Npn \color_math:nn #1#2
39255 {
39256   \__color_math:nn {#2}
39257   { \__color_parse:nN {#1} \l__color_current_tl }
39258 }
39259 \cs_new_protected:Npn \color_math:nnn #1#2#3
39260 {
39261   \__color_math:nn {#3}
39262   { \__color_select_main:Nnn \l__color_current_tl {#1} {#2} }
39263 }
39264 \cs_new_protected:Npn \__color_math:nn #1#2
39265 {
39266   \seq_gpush:NV \g__color_math_seq \l__color_current_tl
39267   #2
39268   \__color_select_math:N \l__color_current_tl
39269   #1
39270   \__color_math_scan:w
39271 }

```

(End of definition for `\color_math:nn`, `\color_math:nnn`, and `\__color_math:nn`. These functions are documented on page 336.)

`\__color_math_scan:w` The complication when changing the color back is due to the fact that the `\color_-`
`\__color_math_scan_auxi:` `math:nn(n)` may be followed by `^` or `_` or the hidden superscript (for example `'`) and its
`\__color_math_scan_auxii:` argument may end in a `\mathop` in which case the sub- and superscripts may be attached
`\__color_math_scan_end:` as `\limits` instead of after the material. All cases need separate treatment. To avoid repeatedly collecting the same token, we first check for an alignment tab: assuming we

don't have one of those, we can “recycle” `\l_peek_token` safely. As we have an explicit `\c_alignment_token`, there needs to be an align-safe group present.

```

39272 \cs_new_protected:Npn \__color_math_scan:w
39273 {
39274   \peek_remove_filler:n
39275   {
39276     \group_align_safe_begin:
39277     \peek_catcode:NTF \c_alignment_token
39278     {
39279       \group_align_safe_end:
39280       \__color_math_scan_end:
39281     }
39282     {
39283       \group_align_safe_end:
39284       \__color_math_scan_auxi:
39285     }
39286   }
39287 }

```

Dealing with literal `_` and `^` is easy, and as we have exactly two cases, we can hard-code this. We use a hard-coded list for limits: these are all primitives. The `\use_none:n` here also removes the test token so it is left just in the right place.

```

39288 \cs_new_protected:Npn \__color_math_scan_auxi:
39289 {
39290   \token_case_catcode:NnTF \l_peek_token
39291   {
39292     \c_math_subscript_token { }
39293     \c_math_superscript_token { }
39294   }
39295   { \__color_math_scripts:Nw }
39296   {
39297     \token_case_meaning:NnTF \l_peek_token
39298     {
39299       \tex_limits:D { \tex_limits:D }
39300       \tex_nolimits:D { \tex_nolimits:D }
39301       \tex_displaylimits:D { \tex_displaylimits:D }
39302     }
39303     { \__color_math_scan:w \use_none:n }
39304     { \__color_math_scan_auxii: }
39305   }
39306 }

```

The one final case to handle is math-active tokens, most obviously `'`, as these won't be covered earlier.

```

39307 \cs_new_protected:Npn \__color_math_scan_auxii:
39308 {
39309   \tl_map_inline:Nn \l_color_math_active_tl
39310   {
39311     \token_if_eq_meaning:NNT \l_peek_token ##1
39312     {
39313       \tl_map_break:n
39314       {
39315         \use_i:nn
39316         { \__color_math_scan_auxiii:N ##1 }

```

```

39317         }
39318     }
39319     \_color_math_scan_end:
39320 }
39321 }
39322 \cs_new_protected:Npn \_color_math_scan_auxiii:N #1
39323 {
39324     \exp_after:wN \exp_after:wN \exp_after:wN \_color_math_scan:w
39325     \char_generate:nn { '#1 } { 13 }
39326 }
39327 \cs_new_protected:Npn \_color_math_scan_end:
39328 {
39329     \_color_backend_reset:
39330     \seq_gpop:NN \g_color_math_seq \l_color_current_tl
39331 }

```

(End of definition for `\_color_math_scan:w` and others.)

`\_color_math_scripts:Nw` The tricky part of handling sub and superscripts is that we have to reset color to the one that is on the stack but reset it back to what it was before to allow for cases like

`\[ \color{math:n}{red}{a + \sum}_{i=1}^n \]`

Here, $\TeX$ constructs a `\vbox` stacking subscript, summation sign, and superscript. So technically the superscript comes first and the `\sum` that should get colored red is the middle.

The approach here is to set up a brace group immediately after the script token, then to set the color appropriately in that argument. We need an extra group to keep the color contained, and as we need to allow for an explicit closing brace in the source, the inner group also is a brace one rather than `\group_begin:-`based. At the end of the outer group we need to insert `\_color_math_scan:w` to continue the search for a second script token.

Notice that here we *don't* need to use the math-specific color selector as we can allow the `\group_insert_after:N \@_backend_reset:` to operate normally.

```

39332 \cs_new_protected:Npn \_color_math_scripts:Nw #1
39333 {
39334     #1
39335     \c_group_begin_token
39336     \c_group_begin_token
39337     \seq_get:NN \g_color_math_seq \l_color_current_tl
39338     \_color_select:N \l_color_current_tl
39339     \group_insert_after:N \c_group_end_token
39340     \group_insert_after:N \_color_math_scan:w
39341     \peek_remove_filler:n
39342     {
39343         \peek_catcode_remove:NF \c_group_begin_token
39344         { \_color_math_script_aux:N }
39345     }
39346 }

```

Deal with the case where we do not have an explicit brace pair in the source.

```

39347 \cs_new_protected:Npn \_color_math_script_aux:N #1 { #1 \c_group_end_token }

```

(End of definition for `\_color_math_scripts:Nw` and `\_color_math_script_aux:N`.)

99.9 Fill and stroke color

```

\color_fill:n
\color_stroke:n
\color_fill:nn
\color_stroke:nn
\__color_draw:nnn
39348 \cs_new_protected:Npn \color_fill:n #1
39349 {
39350   \__color_parse:nN {#1} \l__color_current_tl
39351   \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
39352 }
39353 \cs_new_protected:Npn \color_stroke:n #1
39354 {
39355   \__color_parse:nN {#1} \l__color_current_tl
39356   \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
39357 }
39358 \cs_new_protected:Npn \color_fill:nn #1#2
39359 {
39360   \__color_select_main:Nnn \l__color_current_tl {#1} {#2}
39361   \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
39362 }
39363 \cs_new_protected:Npn \color_stroke:nn #1#2
39364 {
39365   \__color_select_main:Nnn \l__color_current_tl {#1} {#2}
39366   \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
39367 }
39368 \cs_new_protected:Npn \__color_draw:nnn #1#2#3
39369 {
39370   \use:c { __color_backend_ #3 _ #1 :n } {#2}
39371   \exp_args:Nc \group_insert_after:N { __color_backend_ #3 _ reset: }
39372 }

```

(End of definition for `\color_fill:n` and others. These functions are documented on page 335.)

99.10 Defining named colors

`\l__color_named_tl` Space to store the detail of the named color.

```

39373 \tl_new:N \l__color_named_tl

```

(End of definition for `\l__color_named_tl`.)

```

\color_set:nn
\__color_set:nnn
\__color_set:nn
\__color_set:nnw
\color_set:nnn
\__color_set_aux:nnn
\__color_set_colon:nnw
\__color_set_loop:nw
\color_set_eq:nn
237 Defining named colors means working through the model list and saving both the “main”
238 color and any equivalents in other models. Even if there is only one model, we store a
239 prop as well as a tl, as there could be grouping weirdness, etc. When setting using an
240 expression, we need to avoid any fixed model issues, which is done without a group as in
241 l3keys.
39374 \cs_new_protected:Npn \color_set:nn #1#2
39375 {
39376   \exp_args:NV \__color_set:nnn
39377   \l_color_fixed_model_tl {#1} {#2}
39378 }
39379 \cs_new_protected:Npn \__color_set:nnn #1#2#3
39380 {
39381   \tl_clear:N \l_color_fixed_model_tl
39382   \__color_set:nn {#2} {#3}
39383   \tl_set:Nn \l_color_fixed_model_tl {#1}

```

```

39384 }
39385 \cs_new_protected:Npn \__color_set:nn #1#2
39386 {
39387   \str_if_eq:nnF {#1} { . }
39388   {
39389     \__color_parse:nN {#2} \l__color_named_tl
39390     \tl_clear_new:c { l__color_named_ #1 _tl }
39391     \tl_set:ce { l__color_named_ #1 _tl }
39392       { \__color_model:N \l__color_named_tl }
39393     \prop_clear_new:c { l__color_named_ #1 _prop }
39394     \prop_put:cve { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
39395       { \__color_values:N \l__color_named_tl }
39396     \__color_set:nnw {#1} {#2} #2 ! \s__color_stop
39397   }
39398 }

```

When setting an expression-based color, there could be multiple model data available for one or more of the input colors. Where that is true for the *first* named color in an expression, we re-parse the expression when they are also parameter-based: only **cm**yk, **gray** and **rgb** make any sense here. There is a bit of a performance hit but this should be rare and taking place during set-up.

```

39399 \cs_new_protected:Npn \__color_set:nnw #1#2#3 ! #4 \s__color_stop
39400 {
39401   \clist_map_inline:nn { cmyk , gray , rgb }
39402   {
39403     \prop_get:cnNT { l__color_named_ #3 _prop } {##1} \l__color_tmp_tl
39404     {
39405       \prop_if_in:cnF { l__color_named_ #1 _prop } {##1}
39406       {
39407         \group_begin:
39408           \bool_set_true:N \l__color_ignore_error_bool
39409           \tl_set:cn { l__color_named_ #3 _tl } {##1}
39410           \__color_parse:nN {#2} \l__color_tmp_tl
39411           \exp_args:NNNV \group_end:
39412           \tl_set:Nn \l__color_tmp_tl \l__color_tmp_tl
39413           \prop_put:cee { l__color_named_ #1 _prop }
39414             { \__color_model:N \l__color_tmp_tl }
39415             { \__color_values:N \l__color_tmp_tl }
39416         }
39417       }
39418     }
39419   }
39420 \cs_new_protected:Npn \color_set:nnn #1#2#3
39421 {
39422   \str_if_eq:nnF {#1} { . }
39423   {
39424     \tl_clear_new:c { l__color_named_ #1 _tl }
39425     \prop_clear_new:c { l__color_named_ #1 _prop }
39426     \exp_args:Ne \__color_set_aux:nnn { \tl_to_str:n {#2} }
39427       {#1} {#3}
39428   }
39429 }
39430 \cs_new_protected:Npn \__color_set_aux:nnn #1#2#3
39431 {

```

```

39432 \exp_not:N \__color_set_colon:nnw {#2} {#3}
39433 #1 \c_colon_str \c_colon_str \exp_not:N \s__color_stop
39434 }
39435 \use:e
39436 {
39437 \cs_new_protected:Npn \exp_not:N \__color_set_colon:nnw
39438 #1#2 #3 \c_colon_str #4 \c_colon_str
39439 #5 \exp_not:N \s__color_stop
39440 }
39441 {
39442 \tl_if_blank:nTF {#4}
39443 { \__color_set_loop:nw {#1} #3 }
39444 { \__color_set_loop:nw {#1} #4 }
39445 / / \s__color_mark #2 / / \s__color_stop
39446 }
39447 \cs_new_protected:Npn \__color_set_loop:nw
39448 #1#2 / #3 \s__color_mark #4 / #5 \s__color_stop
39449 {
39450 \tl_if_blank:nF {#2}
39451 {
39452 \__color_select:nnN {#2} {#4} \l__color_named_tl
39453 \tl_set:Nx \l__color_tmp_tl { \__color_model:N \l__color_named_tl }
39454 \tl_if_empty:cT { \l__color_named_ #1 _tl }
39455 { \tl_set_eq:cN { \l__color_named_ #1 _tl } \l__color_tmp_tl }
39456 \prop_put:cVe { \l__color_named_ #1 _prop } \l__color_tmp_tl
39457 { \__color_values:N \l__color_named_tl }
39458 \__color_set_loop:nw {#1} #3 \s__color_mark #5 \s__color_stop
39459 }
39460 }
39461 \cs_new_protected:Npn \color_set_eq:nn #1#2
39462 {
39463 \color_if_exist:nTF {#2}
39464 {
39465 \tl_clear_new:c { \l__color_named_ #1 _tl }
39466 \prop_clear_new:c { \l__color_named_ #1 _prop }
39467 \str_if_eq:nnTF {#2} { . }
39468 {
39469 \tl_set:ce { \l__color_named_ #1 _tl }
39470 { \__color_model:N \l__color_current_tl }
39471 \prop_put:cve { \l__color_named_ #1 _prop } { \l__color_named_ #1 _tl }
39472 { \__color_values:N \l__color_current_tl }
39473 }
39474 {
39475 \tl_set_eq:cc { \l__color_named_ #1 _tl } { \l__color_named_ #2 _tl }
39476 \prop_set_eq:cc { \l__color_named_ #1 _prop } { \l__color_named_ #2 _prop }
39477 }
39478 }
39479 {
39480 \msg_error:nnn { color } { unknown-color } {#2}
39481 }
39482 }

```

(End of definition for `\color_set:nn` and others. These functions are documented on page 334.)

A small set of colors are always defined.

```

39483 \color_set:nnn { black } { gray } { 0 }
39484 \color_set:nnn { white } { gray } { 1 }
39485 \color_set:nnn { cyan } { cmyk } { 1 , 0 , 0 , 0 }
39486 \color_set:nnn { magenta } { cmyk } { 0 , 1 , 0 , 0 }
39487 \color_set:nnn { yellow } { cmyk } { 0 , 0 , 1 , 0 }
39488 \color_set:nnn { red } { rgb } { 1 , 0 , 0 }
39489 \color_set:nnn { green } { rgb } { 0 , 1 , 0 }
39490 \color_set:nnn { blue } { rgb } { 0 , 0 , 1 }

```

`\l__color_named_._prop`
`\l__color_named_._tl`

A special named color: this is always defined though not fixed in definition.

```

39491 \prop_new:c { l__color_named_._prop }
39492 \tl_new:c { l__color_named_._tl }
39493 \tl_set:ce { l__color_named_._tl } { \__color_model:N \l__color_current_tl }
39494 \prop_put:cve { l__color_named_._prop } { l__color_named_._tl }
39495 { \__color_values:N \l__color_current_tl }

```

(End of definition for `\l__color_named_._prop` and `\l__color_named_._tl`.)

99.11 Exporting colors

`\color_export:nnN`
`\color_export:nnnN`
`\__color_export:nN`
`\__color_export:nnnN`

```

39496 \cs_new_protected:Npn \color_export:nnN #1#2#3
39497 {
39498   \group_begin:
39499     \tl_if_exist:cT { c__color_export_ #2 _tl }
39500     { \tl_set_eq:Nc \l_color_fixed_model_tl { c__color_export_ #2 _tl } }
39501     \__color_parse:nN {#1} #3
39502     \__color_export:nN {#2} #3
39503     \exp_args:NNNV \group_end:
39504     \tl_set:Nn #3 #3
39505   }
39506 \cs_new_protected:Npn \color_export:nnnN #1#2#3#4
39507 {
39508   \__color_select_main:Nnn #4 {#1} {#2}
39509   \__color_export:nN {#3} #4
39510 }
39511 \cs_new_protected:Npn \__color_export:nN #1#2
39512 { \exp_after:wN \__color_export:nnnN #2 {#1} #2 }
39513 \cs_new:Npn \__color_export:nnnN #1#2#3#4
39514 {
39515   \cs_if_exist_use:cF { __color_export_format_ #3 :nnN }
39516   {
39517     \msg_error:nnn { color } { unknown-export-format } {#3}
39518     \use_none:nnn
39519   }
39520   {#1} {#2} #4
39521 }

```

(End of definition for `\color_export:nnN` and others. These functions are documented on page 337.)

`\__color_export_format_backend:nnN`

Simple.

```

39522 \cs_new_protected:Npn \__color_export_format_backend:nnN #1#2#3
39523 { \tl_set:Nn #3 { {#1} {#2} } }

```


(End of definition for _color_export_format_backend:nnN.)

_color_export:nnnNN A generic auxiliary for cases where only one model is appropriate.

```

39524 \cs_new_protected:Npn \_color_export:nnnNN #1#2#3#4#5
39525 {
39526   \str_if_eq:nnTF {#2} {#1}
39527   { #5 #4 #3 \s_color_stop }
39528   {
39529     \_color_convert:nnnN {#2} {#1} {#3} #4
39530     \exp_after:wN #5 \exp_after:wN #4
39531     #4 \s_color_stop
39532   }
39533 }

```

(End of definition for _color_export:nnnNN.)

```

\_color_export_comma-sep-cmyk_tl
\_color_export_comma-sep-rgb_tl
\_color_export_HTML_tl
\_color_export_space-sep-cmyk_tl
\_color_export_space-sep-rgb_tl
39534 \tl_const:cn { c_color_export_comma-sep-cmyk_tl } { cmyk }
39535 \tl_const:cn { c_color_export_comma-sep-rgb_tl } { rgb }
39536 \tl_const:Nn \_color_export_HTML_tl { rgb }
39537 \tl_const:cn { c_color_export_space-sep-cmyk_tl } { cmyk }
39538 \tl_const:cn { c_color_export_space-sep-rgb_tl } { rgb }

```

(End of definition for _color_export_comma-sep-cmyk_tl and others.)

```

\_color_export_format_comma-sep-cmyk:nnN
\_color_export_format_comma-sep-rgb:nnN
\_color_export_format_space-sep-cmyk:nnN
\_color_export_format_space-sep-rgb:nnN
39539 \group_begin:
39540 \cs_set_protected:Npn \_color_tmp:w #1#2
39541 {
39542   \cs_new_protected:cpe { _color_export_format_ #1 :nnN } ##1##2##3
39543   {
39544     \exp_not:N \_color_export:nnnNN {#2} {##1} {##2} ##3
39545     \exp_not:c { _color_export_ #1 :Nw }
39546   }
39547 }
39548 \_color_tmp:w { comma-sep-cmyk } { cmyk }
39549 \_color_tmp:w { comma-sep-rgb } { rgb }
39550 \_color_tmp:w { HTML } { rgb }
39551 \_color_tmp:w { space-sep-cmyk } { cmyk }
39552 \_color_tmp:w { space-sep-rgb } { rgb }
39553 \group_end:

```

(End of definition for _color_export_format_comma-sep-cmyk:nnN and others.)

```

\_color_export_space-sep-cmyk:Nw
\_color_export_comma-sep-cmyk:Nw
39554 \cs_new_protected:cpn { _color_export_comma-sep-cmyk:Nw }
39555   #1#2 ~ #3 ~ #4 ~ #5 \s_color_stop
39556   { \tl_set:Nn #1 { #2 , #3 , #4 , #5 } }
39557 \cs_new_protected:cpn { _color_export_space-sep-cmyk:Nw } #1#2 \s_color_stop
39558   { \tl_set:Nn #1 {#2} }

```

(End of definition for _color_export_space-sep-cmyk:Nw and _color_export_comma-sep-cmyk:Nw.)

```

\__color_export_comma-sep-rgb:Nw
\__color_export_HTML:Nw
\__color_export_space-sep-rgb:Nw
\__color_export_HTML:n
39559 \cs_new_protected:cpn { __color_export_comma-sep-rgb:Nw } #1#2 ~ #3 ~ #4 \s__color_stop
39560 { \tl_set:Nc #1 { #2 , #3 , #4 } }
39561 \cs_new_protected:Npn \__color_export_HTML:Nw #1#2 ~ #3 ~ #4 \s__color_stop
39562 {
39563   \tl_set:Nc #1
39564   {
39565     \__color_export_HTML:n {#2}
39566     \__color_export_HTML:n {#3}
39567     \__color_export_HTML:n {#4}
39568   }
39569 }
39570 \cs_new:Npn \__color_export_HTML:n #1
39571 {
39572   \fp_compare:nNnTF {#1} = { 0 }
39573   { 00 }
39574   {
39575     \fp_compare:nNnT { #1 * 255 } < { 16 } { 0 }
39576     \int_to_Hex:n { \fp_to_int:n { #1 * 255 } }
39577   }
39578 }
39579 \cs_new_protected:cpn { __color_export_space-sep-rgb:Nw } #1#2 \s__color_stop
39580 { \tl_set:Nn #1 {#2} }

```

(End of definition for __color_export_comma-sep-rgb:Nw and others.)

99.12 Additional color models

```

\l__color_tmp_prop
39581 \prop_new:N \l__color_tmp_prop

```

(End of definition for \l__color_tmp_prop.)

`\g__color_model_int` A tracker for the total number of new models.

```

39582 \int_new:N \g__color_model_int

```

(End of definition for \g__color_model_int.)

`\c__color_fallback_cmyk_tl` `\c__color_fallback_gray_tl` `\c__color_fallback_rgb_tl` For every colorspace, we define one of the base colorspaces as a fallback. The base colorspaces themselves are their own fallback.

```

39583 \tl_const:Nn \c__color_fallback_cmyk_tl { cmyk }
39584 \tl_const:Nn \c__color_fallback_gray_tl { gray }
39585 \tl_const:Nn \c__color_fallback_rgb_tl { rgb }

```

(End of definition for \c__color_fallback_cmyk_tl, \c__color_fallback_gray_tl, and \c__color_fallback_rgb_tl.)

`\g__color_colorants_prop` Mapping from names to colorants.

```

39586 \prop_new:N \g__color_colorants_prop
39587 \prop_gput:Nnn \g__color_colorants_prop { black } { Black }
39588 \prop_gput:Nnn \g__color_colorants_prop { blue } { Blue }
39589 \prop_gput:Nnn \g__color_colorants_prop { cyan } { Cyan }
39590 \prop_gput:Nnn \g__color_colorants_prop { green } { Green }

```

```

39591 \prop_gput:Nnn \g__color_colorants_prop { magenta } { Magenta }
39592 \prop_gput:Nnn \g__color_colorants_prop { none } { None }
39593 \prop_gput:Nnn \g__color_colorants_prop { red } { Red }
39594 \prop_gput:Nnn \g__color_colorants_prop { yellow } { Yellow }

```

(End of definition for `\g__color_colorants_prop`.)

```

\c__color_model_whitepoint_CIELAB_a_tl
\c__color_model_whitepoint_CIELAB_b_tl
\c__color_model_whitepoint_CIELAB_e_tl
\c__color_model_whitepoint_CIELAB_d50_tl
\c__color_model_whitepoint_CIELAB_d55_tl
\c__color_model_whitepoint_CIELAB_d65_tl
\c__color_model_whitepoint_CIELAB_d75_tl

```

Whitepoint data for the CIELAB profiles.

```

39595 \tl_const:Nn \c__color_model_whitepoint_CIELAB_a_tl { 1.0985 ~ 1 ~ 0.3558 }
39596 \tl_const:Nn \c__color_model_whitepoint_CIELAB_b_tl { 0.9807 ~ 1 ~ 1.1822 }
39597 \tl_const:Nn \c__color_model_whitepoint_CIELAB_e_tl { 1 ~ 1 ~ 1 }
39598 \tl_const:cn { c__color_model_whitepoint_CIELAB_d50_tl } { 0.9642 ~ 1 ~ 0.8251 }
39599 \tl_const:cn { c__color_model_whitepoint_CIELAB_d55_tl } { 0.9568 ~ 1 ~ 0.9214 }
39600 \tl_const:cn { c__color_model_whitepoint_CIELAB_d65_tl } { 0.9504 ~ 1 ~ 1.0888 }
39601 \tl_const:cn { c__color_model_whitepoint_CIELAB_d75_tl } { 0.9497 ~ 1 ~ 1.2261 }

```

(End of definition for `\c__color_model_whitepoint_CIELAB_a_tl` and others.)

```
\c__color_model_range_CIELAB_tl
```

The range for CIELAB color spaces.

```
39602 \tl_const:Nn \c__color_model_range_CIELAB_tl { 0 ~ 100 ~ -128 ~ 127 ~ -128 ~ 127 }
```

(End of definition for `\c__color_model_range_CIELAB_tl`.)

```
\g__color_alternative_model_prop
```

For tracking the alternative model set up for separations, etc.

```

39603 \prop_new:N \g__color_alternative_model_prop
39604 \clist_map_inline:nn { cyan , magenta , yellow , black }
39605 { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { cmyk } }
39606 \clist_map_inline:nn { red , green , blue }
39607 { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { rgb } }

```

(End of definition for `\g__color_alternative_model_prop`.)

```
\g__color_alternative_values_prop
```

Same for the values: a bit more involved.

```

39608 \prop_new:N \g__color_alternative_values_prop
39609 \prop_gput:Nnn \g__color_alternative_values_prop { cyan } { 1 , 0 , 0 , 0 }
39610 \prop_gput:Nnn \g__color_alternative_values_prop { magenta } { 0 , 1 , 0 , 0 }
39611 \prop_gput:Nnn \g__color_alternative_values_prop { yellow } { 0 , 0 , 1 , 0 }
39612 \prop_gput:Nnn \g__color_alternative_values_prop { black } { 0 , 0 , 0 , 1 }
39613 \prop_gput:Nnn \g__color_alternative_values_prop { red } { 1 , 0 , 0 }
39614 \prop_gput:Nnn \g__color_alternative_values_prop { green } { 0 , 1 , 0 }
39615 \prop_gput:Nnn \g__color_alternative_values_prop { blue } { 0 , 0 , 1 }

```

(End of definition for `\g__color_alternative_values_prop`.)

```
\color_model_new:nnn
```

Set up a new model: in general this has to be handled by a family-dependent function.

```
\__color_model_new:nnn
```

To avoid some “interesting” questions with casing, we fold the case of the family name. The key–value list should always be present, so we convert it up-front to a `prop`, then deal with the detail on a per-family basis.

```

39616 \cs_new_protected:Npn \color_model_new:nnn #1#2#3
39617 {
39618   \exp_args:Nee \__color_model_new:nnn
39619   { \tl_to_str:n {#1} }
39620   { \str_casefold:n {#2} } {#3}
39621 }
39622 \cs_new_protected:Npn \__color_model_new:nnn #1#2#3

```

```

39623 {
39624   \cs_if_exist:cTF { __color_parse_model_ #1 :w }
39625   {
39626     \msg_error:nnn { color } { model-already-defined } {#1}
39627   }
39628   {
39629     \cs_if_exist:cTF { __color_model_ #2 :n }
39630     {
39631       \prop_set_from_keyval:Nn \l__color_tmp_prop {#3}
39632       \use:c { __color_model_ #2 :n } {#1}
39633     }
39634     {
39635       \msg_error:nnn { color } { unknown-model-type } {#2}
39636     }
39637   }
39638 }

```

(End of definition for \color_model_new:nnn and __color_model_new:nnn. This function is documented on page 337.)

__color_model_init:nnn A shared auxiliary to do the basics of setting up a new model: reserve a number, create a white-equivalent, set up links to the backend.

```

39639 \cs_new_protected:Npn \__color_model_init:nnn #1#2#3
39640 {
39641   \int_gincr:N \g__color_model_int
39642   \clist_map_inline:nn { fill , stroke , select }
39643   {
39644     \cs_new_protected:cpe { __color_backend_ ##1 _ #1 :n } ####1
39645     {
39646       \exp_not:c { __color_backend_ ##1 _ #2 :nn }
39647       { color \int_use:N \g__color_model_int } {####1}
39648     }
39649   }
39650   \cs_new_protected:cpe { __color_model_ #1 _white: }
39651   {
39652     \prop_put:Nnn \exp_not:N \l__color_named_white_prop {#1}
39653     { \exp_not:n {#3} }
39654     \exp_not:N \int_compare:nNnF { \tex_currentgrouplevel:D } = 0
39655     { \group_insert_after:N \exp_not:c { __color_model_ #1 _ white: } }
39656   }
39657   \use:c { __color_model_ #1 _white: }
39658 }
39659 \cs_generate_variant:Nn \__color_model_init:nnn { nne }

```

(End of definition for __color_model_init:nnn.)

__color_model_separation:n Separations must have a “real” name, which is pretty easy to find.

```

\__color_model_separation:nn 39660 \cs_new_protected:Npn \__color_model_separation:n #1
\__color_model_separation:nnn 39661 {
\__color_model_separation:w 39662   \prop_get:NnNTF \l__color_tmp_prop { name }
\__color_model_separation_cmyk:nnnnnn 39663   \l__color_tmp_tl
\__color_model_separation_gray:nnnnnn 39664   {
\__color_model_separation_rgb:nnnnnn 39665     \exp_args:NV \__color_model_separation:nn
\__color_model_convert:nnn 39666     \l__color_tmp_tl {#1}
\__color_model_separation_CIELAB:nnnnnn 39667   }
\__color_model_separation_CIELAB:nnnnnn

```

```

39668     {
39669         \msg_error:nnn { color }
39670         { separation-requires-name } {#1}
39671     }
39672 }

```

We have two keys to find at this stage: the alternative space model and linked values.

```

39673 \cs_new_protected:Npn \__color_model_separation:nn #1#2
39674 {
39675     \prop_get:NnNTF \l__color_tmp_prop { alternative-model }
39676     \l__color_tmp_tl
39677     {
39678         \exp_args:NV \__color_model_separation:nnn
39679         \l__color_tmp_tl {#2} {#1}
39680     }
39681     {
39682         \msg_error:nnn { color }
39683         { separation-alternative-model } {#2}
39684     }
39685 }
39686 \cs_new_protected:Npn \__color_model_separation:nnn #1#2#3
39687 {
39688     \cs_if_exist:cTF { __color_model_separation_ #1 :nnnnnn }
39689     {
39690         \prop_get:NnNTF \l__color_tmp_prop { alternative-values }
39691         \l__color_tmp_tl
39692         {
39693             \exp_after:wN \__color_model_separation:w \l__color_tmp_tl
39694             , 0 , 0 , 0 , 0 \s__color_stop {#2} {#3} {#1}
39695         }
39696         {
39697             \msg_error:nnn { color }
39698             { separation-alternative-values } {#2}
39699         }
39700     }
39701     {
39702         \msg_error:nnn { color }
39703         { unknown-alternative-model } {#1}
39704     }
39705 }

```

As each alternative space leads to a different requirement for conversion, and as there are only a small number of choices, we manually split the data and then set up. Notice that mixing tints is really just the same as mixing **gray**. The **white** color is special, as it allows tints to be adjusted without an additional color space. To make sure the data is set for that at all group levels, we need to work on a per-level basis. Within the output, only the set-up needs the “real” name of the colorspace: we use a simple tracking number for general usage as this is a clear namespace without issues of escaping chars.

```

39706 \cs_new_protected:Npn \__color_model_separation:w
39707     #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7#8
39708 {
39709     \__color_model_init:nnn {#6} { separation } { 0 }
39710     \cs_new_eq:cN { __color_parse_mix_ #6 :nw } \__color_parse_mix_gray:nw
39711     \cs_new:cpn { __color_parse_model_ #6 :w } ##1 , ##2 \s__color_stop

```

```

39712     { {#6} { \__color_parse_number:n {##1} } }
39713 \use:c { __color_model_separation_ #8 :nnnnnn }
39714     {#6} {#7} {#1} {#2} {#3} {#4}
39715 \prop_gput:Nnn \g__color_alternative_model_prop {#6} {#8}
39716 \prop_gput:Nne \g__color_colorants_prop {#6}
39717     { \str_convert_pdfname:n {#7} }
39718 }
39719 \cs_new_protected:Npn \__color_model_separation_cmyk:nnnnnn #1#2#3#4#5#6
39720 {
39721     \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
39722     \cs_new:cpn { __color_convert_ #1 _cmyk:w } ##1 \s__color_stop
39723     {
39724         \fp_eval:n {##1 * #3} ~
39725         \fp_eval:n {##1 * #4} ~
39726         \fp_eval:n {##1 * #5} ~
39727         \fp_eval:n {##1 * #6}
39728     }
39729     \cs_new:cpn { __color_convert_cmyk_ #1 :w } ##1 \s__color_stop { 1 }
39730     \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 , #6 }
39731     \__color_backend_separation_init:nnnnn {#2} { /DeviceCMYK } { }
39732     { 0 ~ 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 ~ #6 }
39733 }
39734 \cs_new_protected:Npn \__color_model_separation_rgb:nnnnnn #1#2#3#4#5#6
39735 {
39736     \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
39737     \cs_new:cpn { __color_convert_ #1 _rgb:w } ##1 \s__color_stop
39738     {
39739         \fp_eval:n {##1 * #3} ~
39740         \fp_eval:n {##1 * #4} ~
39741         \fp_eval:n {##1 * #5}
39742     }
39743     \cs_new:cpn { __color_convert_rgb_ #1 :w } ##1 \s__color_stop { 1 }
39744     \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 }
39745     \__color_backend_separation_init:nnnnn {#2} { /DeviceRGB } { }
39746     { 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 }
39747 }
39748 \cs_new_protected:Npn \__color_model_separation_gray:nnnnnn #1#2#3#4#5#6
39749 {
39750     \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
39751     \cs_new:cpn { __color_convert_ #1 _gray:w } ##1 \s__color_stop
39752     { \fp_eval:n {##1 * #3} }
39753     \cs_new:cpn { __color_convert_gray_ #1 :w } ##1 \s__color_stop { 1 }
39754     \prop_gput:Nnn \g__color_alternative_values_prop {#1} {#3}
39755     \__color_backend_separation_init:nnnnn {#2} { /DeviceGray } { } { 0 } {#3}
39756 }

```

Generic model conversion *via* an alternative intermediate.

```

39757 \cs_new_protected:Npn \__color_model_convert:nnn #1#2#3
39758 {
39759     \cs_new:cpe { __color_convert_ #1 _ #3 :w } ##1 \s__color_stop
39760     {
39761         \exp_not:N \exp_args:NNe \exp_not:N \use:nn
39762         \exp_not:c { __color_convert_ #2 _ #3 :w }
39763         { \exp_not:c { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop }
39764         \c_space_tl \exp_not:N \s__color_stop

```

```

39765     }
39766 }

```

Setting up for CIELAB needs a bit more work: there is the illuminant and the need for an appropriate object.

```

39767 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6
39768 {
39769     \prop_get:NnNF \l__color_tmp_prop { illuminant }
39770     \l__color_tmp_tl
39771     {
39772         \msg_error:nnn { color }
39773         { CIELAB-requires-illuminant } {#1}
39774         \tl_set:Nn \l__color_tmp_tl { d50 }
39775     }
39776     \exp_args:NV \__color_model_separation_CIELAB:nnnnnnn
39777     \l__color_tmp_tl {#1} {#2} {#3} {#4} {#5} {#6}
39778 }

```

If a CIELAB space is being set up, we need the illuminant, then create the appropriate set up. At present, this doesn't include BlackPoint or Range data, but that may be added later. As CIELAB colors cannot be converted to anything else, we fallback to producing black in the gray colorspace: the user should set up a second model for colors set up this way.

```

39779 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnnn #1#2#3#4#5#6#7
39780 {
39781     \tl_if_exist:cTF { c__color_model_whitepoint_CIELAB_ #1 _tl }
39782     {
39783         \__color_backend_separation_init_CIELAB:nnn {#1} {#3} { #4 ~ #5 ~ #6 }
39784         \tl_const:cn { c__color_fallback_ #2 _tl } { gray }
39785         \cs_new:cpn { __color_convert_ #2 _gray:w } ##1 \s__color_stop
39786         { 0 }
39787         \cs_new:cpn { __color_convert_gray_ #2 :w } ##1 \s__color_stop
39788         { 1 }
39789     }
39790     {
39791         \msg_error:nnn { color }
39792         { unknown-CIELAB-illuminant } {#1}
39793     }
39794 }

```

(End of definition for __color_model_separation:n and others.)

```

\__color_model_devicen:n
\__color_model_devicen:nn
\__color_model_devicen:nnn
\__color_model_devicen:nnnn
\__color_model_devicen_parse_1:nn
\__color_model_devicen_parse_2:nn
\__color_model_devicen_parse_3:nn
\__color_model_devicen_parse_4:nn
\__color_model_devicen_parse_generic:nn
\__color_model_devicen_parse:nw
\__color_model_devicen_mix:nw
\__color_model_devicen_init:nnn
\__color_model_devicen_init:nnnn
\__color_model_devicen_transform:w
\__color_model_devicen_transform_1:nnnnn
\__color_model_devicen_transform_3:nnnnn
\__color_model_devicen_transform_4:nnnnn
\__color_model_devicen_transform:nnn
\__color_model_devicen_colorant:n
\__color_model_devicen_convert:nnn
\__color_model_devicen_convert_cmyk:n
\__color_model_devicen_convert_gray:n

```

We require a list of component names here: one might call them colorants, but it's convenient to use T_EX names instead so we slightly adjust the terminology.

```

39795 \cs_new_protected:Npn \__color_model_devicen:n #1
39796 {
39797     \prop_get:NnNTF \l__color_tmp_prop { names }
39798     \l__color_tmp_tl
39799     {
39800         \exp_args:NV \__color_model_devicen:nn
39801         \l__color_tmp_tl {#1}
39802     }
39803     {
39804         \msg_error:nnn { color }
39805         { DeviceN-requires-names } {#1}

```

```

39806     }
39807 }

```

All valid models will have an alternative listed, either hard-coded for the core device ones, or dynamically added for Separations, etc.

```

39808 \cs_new_protected:Npn \__color_model_devicen:nn #1#2
39809 {
39810   \tl_clear:N \l__color_model_tl
39811   \clist_map_inline:nn {#1}
39812   {
39813     \prop_get:NnNTF \g__color_alternative_model_prop {##1}
39814     \l__color_tmp_tl
39815     {
39816       \tl_if_empty:NTF \l__color_model_tl
39817       { \tl_set_eq:NN \l__color_model_tl \l__color_tmp_tl }
39818       {
39819         \str_if_eq:VVF \l__color_model_tl \l__color_tmp_tl
39820         {
39821           \msg_error:nnn { color }
39822             { DeviceN-inconsistent-alternative }
39823             {#2}
39824           \clist_map_break:n { \use_none:nnnn }
39825         }
39826       }
39827     }
39828   {
39829     \str_if_eq:nnF {##1} { none }
39830     {
39831       \msg_error:nnn { color }
39832       { DeviceN-no-alternative }
39833       {#2}
39834     }
39835   }
39836 }
39837 \tl_if_empty:NTF \l__color_model_tl
39838 {
39839   \msg_error:nnn { color }
39840   { DeviceN-no-alternative } {#2}
39841 }
39842 { \exp_args:NV \__color_model_devicen:nnn \l__color_model_tl {#1} {#2} }
39843 }

```

We now complete the data we require by first finding out how many colorants there are, then moving on to begin constructing the function required to map to the alternative color space.

```

39844 \cs_new_protected:Npn \__color_model_devicen:nnn #1#2#3
39845 {
39846   \exp_args:Ne \__color_model_devicen:nnnn
39847   { \clist_count:n {#2} } {#1} {#2} {#3}
39848 }

```

At this stage, we have checked everything is in place, so we can set up the T_EX and backend data structures. As for separations, it’s not really possible in general to have a fallback, so we simply provide “black” for each element.

```

39849 \cs_new_protected:Npn \__color_model_devicen:nnnn #1#2#3#4

```



```

39850 {
39851   \_color_model_init:nne {#4} { devicen }
39852   {
39853     0 \prg_replicate:nn { #1 - 1 } { ~ 0 }
39854   }
39855   \cs_if_exist_use:cF { \_color_model_devicen_parse_ #1 :nn }
39856   { \_color_model_devicen_parse_generic:nn }
39857   {#4} {#1}
39858   \_color_model_devicen_init:nnn {#1} {#2} {#3}
39859   \_color_model_devicen_convert:nnne {#4} {#2} {#3}
39860   {
39861     1 \prg_replicate:nn { #1 - 1 } { ~ 1 }
39862   }
39863 }

```

For short lists of DeviceN colors, we can use hand-tuned parsing. This lines up with other models, where we allow for up to four components. For larger spaces, rather than limit artificially, we use a somewhat slow approach based on open-ended commas-lists.

```

39864 \cs_new_protected:cpn { \_color_model_devicen_parse_1:nn } #1#2
39865 {
39866   \cs_new:cpn { \_color_parse_model_ #1 :w } ##1 , ##2 \s_color_stop
39867   { {#1} { \_color_parse_number:n {##1} } }
39868   \cs_new_eq:cN { \_color_parse_mix_ #1 :nw } \_color_parse_mix_gray:nw
39869 }
39870 \cs_new_protected:cpn { \_color_model_devicen_parse_2:nn } #1#2
39871 {
39872   \cs_new:cpn { \_color_parse_model_ #1 :w } ##1 , ##2 , ##3 \s_color_stop
39873   { {#1} { \_color_parse_number:n {##1} ~ \_color_parse_number:n {##2} } }
39874   \cs_new:cpn { \_color_parse_mix_ #1 :nw }
39875   ##1##2 ~ ##3 \s_color_mark ##4 ~ ##5 \s_color_stop
39876   {
39877     \fp_eval:n { ##2 * ##1 + ##4 * ( 1 - ##1 ) } \c_space_tl
39878     \fp_eval:n { ##3 * ##1 + ##5 * ( 1 - ##1 ) }
39879   }
39880 }
39881 \cs_new_protected:cpn { \_color_model_devicen_parse_3:nn } #1#2
39882 {
39883   \cs_new:cpn { \_color_parse_model_ #1 :w } ##1 , ##2 , ##3 , ##4 \s_color_stop
39884   {
39885     {#1}
39886     {
39887       \_color_parse_number:n {##1} ~
39888       \_color_parse_number:n {##2} ~
39889       \_color_parse_number:n {##3}
39890     }
39891   }
39892   \cs_new_eq:cN { \_color_parse_mix_ #1 :nw } \_color_parse_mix_rgb:nw
39893 }
39894 \cs_new_protected:cpn { \_color_model_devicen_parse_4:nn } #1#2
39895 {
39896   \cs_new:cpn { \_color_parse_model_ #1 :w }
39897   ##1 , ##2 , ##3 , ##4 , ##5 \s_color_stop
39898   {
39899     {#1}

```

```

39900     {
39901         \_color_parse_number:n {##1} ~
39902         \_color_parse_number:n {##2} ~
39903         \_color_parse_number:n {##3} ~
39904         \_color_parse_number:n {##4}
39905     }
39906 }
39907 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \_color_parse_mix_cmyk:nw
39908 }
39909 \cs_new_protected:Npn \_color_model_devicen_parse_generic:nn #1#2
39910 {
39911     \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
39912     {
39913         {#1}
39914         { \_color_model_devicen_parse:nw {#2} ##1 , ##2 , \q_nil , \s__color_stop }
39915     }
39916     \cs_new:cpe { __color_parse_mix_ #1 :nw }
39917     ##1 ##2 \s__color_mark ##3 \s__color_stop
39918     {
39919         \exp_not:N \_color_model_devicen_mix:nw {##1}
39920         ##2 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_mark
39921         ##3 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_stop
39922     }
39923 }
39924 \cs_new:Npn \_color_model_devicen_parse:nw #1#2 , #3 \s__color_stop
39925 {
39926     \int_compare:nNnT {#1} > 0
39927     {
39928         \quark_if_nil:nTF {#2}
39929         { \prg_replicate:nn {#1} { 0 ~ } }
39930         {
39931             \_color_parse_number:n {#2}
39932             \int_compare:nNnT {#1} > 1 { ~ }
39933             \exp_args:Nf \_color_model_devicen_parse:nw
39934             { \int_eval:n { #1 - 1 } } #3 \s__color_stop
39935         }
39936     }
39937 }
39938 \cs_new:Npn \_color_model_devicen_mix:nw #1#2 ~ #3 \s__color_mark #4 ~ #5 \s__color_stop
39939 {
39940     \fp_eval:n { #2 * #1 + #4 * ( 1 - #1 ) }
39941     \quark_if_nil:oF { \tl_head:w #3 \q_stop }
39942     {
39943         \c_space_tl
39944         \_color_model_devicen_mix:nw {#1} #3 \s__color_mark #5 \s__color_stop
39945     }
39946 }

```

To construct the tint transformation, we have to use PostScript. The aim is to have the final tint for each device colorant as

$$1 - \prod_n (1 - X_n D_{X_n})$$

where X is a DeviceN colorant and D is the amount of device colorant that the DeviceN colorant maps to. At the start of the process, the PostScript stack will contain the

X_n values, whilst we have the D values on a per-DeviceN colorant basis. The more convenient approach for us is therefore to take each DeviceN colorant in turn and find the value $1 - X_n D_{X_n}$, multiplying as we go, and finalize with the subtraction. That contrasts to `colorspace`: it splits the process up by process color, which works better when you have a fixed list of colorants. (`colorspace` only supports up to 4 DeviceN colors, and only `cmymk` as the alternative space.) To set this up, we first need to know the number of values in the target color space: this is easily handled as there are a very small range of possibilities. Once we have that information, it's relatively easy to build the required PostScript using some generic code.

```

39947 \cs_new_protected:Npn \__color_model_devicen_init:nnn #1#2#3
39948 {
39949   \exp_args:Ne \__color_model_devicen_init:nnnn
39950   {
39951     \str_case:nn {#2}
39952     {
39953       { cmyk } { 4 }
39954       { gray } { 1 }
39955       { rgb } { 3 }
39956     }
39957   }
39958   {#1} {#2} {#3}
39959 }

```

As we always need to split the alternative values into parts, we use a shared auxiliary and only use a minimal difference between code paths. Construction of the tint transformation is as far as possible done using loops, which means there are some inefficiencies for device colors in the DeviceN space: we roll the stack one-at-a-time even if there is a potential shortcut. However, that way there is nothing to special-case. Once this is sorted, we can write the tint transform object, which will remain as the last object until we sort out the final step: the colorant list.

```

39960 \cs_new_protected:Npn \__color_model_devicen_init:nnnn #1#2#3#4
39961 {
39962   \tl_set:Nx \l__color_tmp_tl
39963   { \prg_replicate:nn {#1} { 1.0 ~ } }
39964   \int_zero:N \l__color_tmp_int
39965   \clist_map_inline:nn {#4}
39966   {
39967     \int_incr:N \l__color_tmp_int
39968     \prop_get:NnN \g__color_alternative_values_prop {##1}
39969     \l__color_value_tl
39970     \exp_after:wN \__color_model_devicen_transform:w
39971     \l__color_value_tl , 0 , 0 , 0 , \s__color_stop {#1} {#2}
39972   }
39973   \tl_put_right:Nx \l__color_tmp_tl
39974   {
39975     \prg_replicate:nn {#1}
39976     { neg ~ 1.0 ~ add ~ #1 ~ -1 ~ roll ~ }
39977     \int_eval:n { #2 + #1 } ~ #1 ~ roll
39978     \prg_replicate:nn {#2} { ~ pop } ~
39979     #1 ~ 1 ~ roll
39980   }
39981   \use:e
39982   {

```

```

39983     \_color_backend_devicen_init:nnn
39984     {
39985         \clist_map_function:nN {#4}
39986         \_color_model_devicen_colorant:n
39987     }
39988     {
39989         \str_case:nn {#3}
39990         {
39991             { cmyk } { /DeviceCMYK }
39992             { gray } { /DeviceGray }
39993             { rgb } { /DeviceRGB }
39994         }
39995     }
39996     { \exp_not:V \l__color_tmp_tl }
39997 }
39998 }
39999 \cs_new_protected:Npn \_color_model_devicen_transform:w
40000 #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7
40001 {
40002     \use:c { __color_model_devicen_transform_ #6 :nnnnn }
40003     {#1} {#2} {#3} {#4} {#7}
40004 }
40005 \cs_new_protected:cpn { __color_model_devicen_transform_1:nnnnn } #1#2#3#4#5
40006 { \_color_model_devicen_transform:nnn {#5} { 1 } {#1} }
40007 \cs_new_protected:cpn { __color_model_devicen_transform_3:nnnnn } #1#2#3#4#5
40008 {
40009     \clist_map_inline:nn { #1 , #2 , #3 }
40010     { \_color_model_devicen_transform:nnn {#5} { 3 } {##1} }
40011 }
40012 \cs_new_protected:cpn { __color_model_devicen_transform_4:nnnnn } #1#2#3#4#5
40013 {
40014     \clist_map_inline:nn { #1 , #2 , #3 , #4 }
40015     { \_color_model_devicen_transform:nnn {#5} { 4 } {##1} }
40016 }
40017 \cs_new_protected:Npn \_color_model_devicen_transform:nnn #1#2#3
40018 {
40019     \tl_put_right:Ne \l__color_tmp_tl
40020     {
40021         \fp_compare:nNnF {#3} = \c_zero_fp
40022         {
40023             \int_eval:n { #1 - \l__color_tmp_int + #2 } ~ index ~
40024             -#3 ~ mul ~ 1.0 ~ add ~ mul ~
40025         }
40026         #2 ~ -1 ~ roll ~
40027     }
40028 }
40029 \cs_new:Npn \_color_model_devicen_colorant:n #1
40030 {
40031     / \prop_item:Nn \g__color_colorants_prop {#1} ~
40032 }

```

Here we need to set up conversion from the DeviceN space to the alternative at the $\text{\TeX}$ level. This also means supplying methods for inter-converting to other parameter-based spaces. Essentially the approach is exactly the same as the PostScript, just expressed in

TeX terms.

```

40033 \cs_new_protected:Npn \__color_model_devicen_convert:nnnn #1#2#3
40034 {
40035   \use:c { __color_model_devicen_convert_ #2 :nnn } {#1} {#3}
40036 }
40037 \cs_generate_variant:Nn \__color_model_devicen_convert:nnnn { nnne }
40038 \cs_new_protected:Npn \__color_model_devicen_convert_cmyk:nnn #1#2
40039 {
40040   \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
40041   \__color_model_devicen_convert:nnnn {#1} { cmyk } { 4 } {#2}
40042 }
40043 \cs_new_protected:Npn \__color_model_devicen_convert_gray:nnn #1#2
40044 {
40045   \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
40046   \__color_model_devicen_convert:nnnn {#1} { gray } { 1 } {#2}
40047 }
40048 \cs_new_protected:Npn \__color_model_devicen_convert_rgb:nnn #1#2
40049 {
40050   \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
40051   \__color_model_devicen_convert:nnnn {#1} { rgb } { 3 } {#2}
40052 }
40053 \cs_new_protected:Npn \__color_model_devicen_convert:nnnnn #1#2#3#4#5
40054 {
40055   \cs_new:cpn { __color_convert_ #2 _ #1 :w } ##1 \s__color_stop {#5}
40056   \cs_new:cpe { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop
40057   {
40058     \exp_not:c { __color_convert_devicen_ #2 : \prg_replicate:nn {#3} { n } w }
40059     \prg_replicate:nn {#3} { { 1 } }
40060     ##1 ~ \exp_not:N \s__color_mark
40061     \clist_map_function:nN {#4} \__color_model_devicen_convert:n
40062     {}
40063     \exp_not:N \s__color_stop
40064   }
40065 }
40066 \cs_new:Npn \__color_model_devicen_convert:n #1
40067 {
40068   {
40069     \exp_args:Ne \__color_model_devicen_convert_aux:n
40070     { \prop_item:Nn \g__color_alternative_values_prop {#1} }
40071   }
40072 }
40073 \cs_new:Npn \__color_model_devicen_convert_aux:n #1
40074 { \__color_model_devicen_convert_aux:w #1 , , , , \s__color_stop }
40075 \cs_new:Npn \__color_model_devicen_convert_aux:w #1 , #2 , #3 , #4 , #5 \s__color_stop
40076 {
40077   {#1}
40078   \tl_if_blank:nF {#2}
40079   {
40080     {#2}
40081     \tl_if_blank:nF {#3}
40082     {
40083       {#3}
40084       \tl_if_blank:nF {#4} { {#4} }
40085     }

```

```

40086     }
40087   }
40088   \cs_new:Npn \__color_convert_devicen_cmyk:nnnnw
40089     #1#2#3#4#5 ~ #6 \s__color_mark #7#8 \s__color_stop
40090   {
40091     \__color_convert_devicen_cmyk:nnnnnnnnn {#5} {#1} {#2} {#3} {#4} #7
40092     #6 \s__color_mark #8 \s__color_stop
40093   }
40094   \cs_new:Npn \__color_convert_devicen_cmyk:nnnnnnnnn #1#2#3#4#5#6#7#8#9
40095   {
40096     \use:e
40097     {
40098       \exp_not:N \__color_convert_devicen_cmyk_aux:nnnnw
40099       { \fp_eval:n { #2 * (1 - (#1 * #6)) } }
40100       { \fp_eval:n { #3 * (1 - (#1 * #7)) } }
40101       { \fp_eval:n { #4 * (1 - (#1 * #8)) } }
40102       { \fp_eval:n { #5 * (1 - (#1 * #9)) } }
40103     }
40104   }
40105   \cs_new:Npn \__color_convert_devicen_cmyk_aux:nnnnw
40106     #1#2#3#4 #5 \s__color_mark #6 \s__color_stop
40107   {
40108     \tl_if_blank:nTF {#5}
40109     {
40110       \fp_eval:n { 1 - #1 } ~
40111       \fp_eval:n { 1 - #2 } ~
40112       \fp_eval:n { 1 - #3 } ~
40113       \fp_eval:n { 1 - #4 }
40114     }
40115     {
40116       \__color_convert_devicen_cmyk:nnnnw {#1} {#2} {#3} {#4}
40117       #5 \s__color_mark #6 \s__color_stop
40118     }
40119   }
40120   \cs_new:Npn \__color_convert_devicen_gray:nw
40121     #1#2 ~ #3 \s__color_mark #4#5 \s__color_stop
40122   {
40123     \__color_convert_devicen_gray:nnn {#2} {#1} #4
40124     #3 \s__color_mark #5 \s__color_stop
40125   }
40126   \cs_new:Npn \__color_convert_devicen_gray:nnn #1#2#3
40127   {
40128     \exp_args:Ne \__color_convert_devicen_gray_aux:nw
40129     { \fp_eval:n { #2 * (1 - (#1 * #3)) } }
40130   }
40131   \cs_new:Npn \__color_convert_devicen_gray_aux:nw
40132     #1 #2 \s__color_mark #3 \s__color_stop
40133   {
40134     \tl_if_blank:nTF {#2}
40135     { \fp_eval:n { 1 - #1 } }
40136     {
40137       \__color_convert_devicen_gray:nw {#1}
40138       #2 \s__color_mark #3 \s__color_stop
40139     }

```

```

40140 }
40141 \cs_new:Npn \__color_convert_devicen_rgb:nnnw
40142   #1#2#3#4 ~ #5 \s__color_mark #6#7 \s__color_stop
40143   {
40144     \__color_convert_devicen_rgb:nnnnnnn {#4} {#1} {#2} {#3} #6
40145     #5 \s__color_mark #7 \s__color_stop
40146   }
40147 \cs_new:Npn \__color_convert_devicen_rgb:nnnnnnn #1#2#3#4#5#6#7
40148   {
40149     \use:e
40150     {
40151       \exp_not:N \__color_convert_devicen_rgb_aux:nnnw
40152         { \fp_eval:n { #2 * (1 - (#1 * #5)) } }
40153         { \fp_eval:n { #3 * (1 - (#1 * #6)) } }
40154         { \fp_eval:n { #4 * (1 - (#1 * #7)) } }
40155     }
40156   }
40157 \cs_new:Npn \__color_convert_devicen_rgb_aux:nnnw
40158   #1#2#3 #4 \s__color_mark #5 \s__color_stop
40159   {
40160     \tl_if_blank:nTF {#4}
40161     {
40162       \fp_eval:n { 1 - #1 } ~
40163       \fp_eval:n { 1 - #2 } ~
40164       \fp_eval:n { 1 - #3 }
40165     }
40166     {
40167       \__color_convert_devicen_rgb:nnnw {#1} {#2} {#3}
40168       #4 \s__color_mark #5 \s__color_stop
40169     }
40170   }

```

(End of definition for __color_model_devicen:n and others.)

\c_color_icc_colorspace_signatures_prop

The signatures in the ICC file header indicating the underlying colorspace. We map it to three values: The number of components, the values corresponding to white, and the range.

```

40171 \prop_const_from_keyval:Nn \c__color_icc_colorspace_signatures_prop
40172   {
40173     % Gray
40174     47524159 = {1} {1} {0} {},
40175     % RGB
40176     52474220 = {3} {0~0~0} {1~1~1} {},
40177     % CMYK
40178     434D594B = {4} {0~0~0~1} {0~0~0~0} {},
40179     % Lab
40180     4C616220 = {3} {0~0~0} {100~0~0} {0~100~-128~127~-128~127}
40181   }

```

(End of definition for \c__color_icc_colorspace_signatures_prop.)

__color_model_iccbased:n
 __color_model_iccbased:nn
 __color_model_iccbased:nnn
 __color_model_iccbased_aux:nnn

For an ICC profile, we need a file name and a number of components. The file name is processed here so the backend can treat it as a string.

```

40182 \cs_new_protected:Npn \__color_model_iccbased:n #1

```

```

40183 {
40184     \prop_get:NnNTF \l__color_tmp_prop { file }
40185     \l__color_tmp_tl
40186     {
40187         \exp_args:NV \__color_model_iccbased:nn
40188         \l__color_tmp_tl {#1}
40189     }
40190     {
40191         \msg_error:nnn { color }
40192         { ICCBased-requires-file } {#1}
40193     }
40194 }
40195 \cs_new_protected:Npn \__color_model_iccbased:nn #1#2
40196 {
40197     \prop_get:NnNTF \c__color_icc_colorspace_signatures_prop
40198     { \file_hex_dump:nnn { #1 } { 17 } { 20 } } \l__color_tmp_tl
40199     {
40200         \exp_last_unbraced:NV \__color_model_iccbased_aux:nnnnnn
40201         \l__color_tmp_tl { #2 } { #1 }
40202     }
40203     {
40204         \msg_error:nnn { color }
40205         { ICCBased-unsupported-colorspace } {#2}
40206     }
40207 }

```

Here, we can use the same internals as for DeviceN approach as we know the number of components. No conversion is possible, so there is no need to worry about that at all.

```

40208 \cs_new_protected:Npn \__color_model_iccbased_aux:nnnnnn #1#2#3#4#5#6
40209 {
40210     \__color_model_init:nnn {#5} { iccbased } {#3}
40211     \tl_const:cn { c__color_fallback_ #5 _tl } { gray }
40212     \cs_new:cpn { __color_convert_ #5 _gray:w } ##1 \s_color_stop { 0 }
40213     \cs_new:cpn { __color_convert_gray_ #5 :w } ##1 \s_color_stop { #2 }
40214     \use:c { __color_model_devicen_parse_ #1 :nn } {#5} {#1}
40215     \exp_args:Ne \__color_backend_iccbased_init:nnn
40216     { \file_full_name:n {#6} } {#1} {#4}
40217 }

```

(End of definition for __color_model_iccbased:n and others.)

99.13 Applying profiles

With a limited range of outcomes, this is largely about getting data to the backend.

```

\color_profile_apply:nn
\__color_profile_apply:nn
    \__color_profile_apply_gray:n
\__color_profile_apply_rgb:n
    \__color_profile_apply_cmyk:n
40218 \cs_new_protected:Npn \color_profile_apply:nn #1#2
40219 {
40220     \exp_args:Ne \__color_profile_apply:nn
40221     { \file_full_name:n {#1} } {#2}
40222 }
40223 \cs_new_protected:Npn \__color_profile_apply:nn #1#2
40224 {
40225     \cs_if_exist_use:cF { __color_profile_apply_ \tl_to_str:n {#2} :n }
40226     {

```



```

40227         \msg_error:nnn { color } { ICC-Device-unknown } {#2}
40228         \use_none:n
40229     }
40230     {#1}
40231 }
40232 \cs_new_protected:Npn \__color_profile_apply_gray:n #1
40233 {
40234     \int_gincr:N \g__color_model_int
40235     \__color_backend_iccbased_device:nnn {#1} { Gray } { 1 }
40236 }
40237 \cs_new_protected:Npn \__color_profile_apply_rgb:n #1
40238 {
40239     \int_gincr:N \g__color_model_int
40240     \__color_backend_iccbased_device:nnn {#1} { RGB } { 3 }
40241 }
40242 \cs_new_protected:Npn \__color_profile_apply_cmyk:n #1
40243 {
40244     \int_gincr:N \g__color_model_int
40245     \__color_backend_iccbased_device:nnn {#1} { CMYK } { 4 }
40246 }

```

(End of definition for `\color_profile_apply:nn` and others. This function is documented on page 338.)

99.14 Diagnostics

```

\color_show:n Extract the information about a color and format for the user: the approach is similar
\color_log:n   to the keys module here.
\__color_show:Nn 40247 \cs_new_protected:Npn \color_show:n
\__color_show:n 40248 { \__color_show:Nn \msg_show:nneeee }
40249 \cs_new_protected:Npn \color_log:n
40250 { \__color_show:Nn \msg_log:nneeee }
40251 \cs_new_protected:Npn \__color_show:Nn #1#2
40252 {
40253     #1 { color } { show }
40254     {#2}
40255     {
40256         \color_if_exist:nT {#2}
40257         {
40258             \exp_args:Nv \__color_show:n { l__color_named_ #2 _tl }
40259             \prop_map_function:cN
40260             { l__color_named_ #2 _prop }
40261             \msg_show_item_unbraced:nn
40262         }
40263     }
40264     { }
40265     { }
40266 }
40267 \cs_new:Npn \__color_show:n #1
40268 {
40269     \msg_show_item_unbraced:nn { model } {#1}
40270 }

```

(End of definition for `\color_show:n` and others. These functions are documented on page 334.)

99.15 Messages

```
40271 \msg_new:nnnn { color } { CIELAB-requires-illuminant }
40272 { CIELAB~color~space~'#1'~require~an~illuminant. }
40273 {
40274   LaTeX~has~been~asked~to~create~a~separation~color~space~using~
40275   CIELAB~specifications,~but~no~\\ \\
40276   \iow_indent:n { illuminant=~<basis> }
40277   \\ \\
40278   key~was~given~with~the~correct~information.~LaTeX~will~use~illuminant~
40279   'd50'~for~recovery.
40280 }
40281 \msg_new:nnnn { color } { conversion-not-available }
40282 { No~model~conversion~available~from~'#1'~to~'#2'. }
40283 {
40284   LaTeX~has~been~asked~to~convert~a~color~from~model~'#1'~
40285   to~model~'#2',~but~there~is~no~method~available~to~do~that.
40286 }
40287 \msg_new:nnnn { color } { DeviceN-inconsistent-alternative }
40288 { DeviceN~color~spaces~require~a~single~alternative~space. }
40289 {
40290   LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
40291   but~the~constituent~colors~do~not~have~a~common~alternative~
40292   color.
40293 }
40294 \msg_new:nnnn { color } { DeviceN-no-alternative }
40295 { DeviceN~color~spaces~require~an~alternative~space. }
40296 {
40297   LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
40298   but~the~constituent~colors~do~not~all~have~a~device~based~alternative.
40299 }
40300 \msg_new:nnnn { color } { DeviceN-requires-names }
40301 { DeviceN~color~space~'#1'~require~a~list~of~names. }
40302 {
40303   LaTeX~has~been~asked~to~create~a~DeviceN~color~space,~
40304   but~no~\\ \\
40305   \iow_indent:n { names=~<names> }
40306   \\ \\
40307   key~was~given~with~the~correct~information.
40308 }
40309 \msg_new:nnnn { color } { ICC-Device-unknown }
40310 { Unknown~device~color~space~'#1'. }
40311 {
40312   LaTeX~has~been~asked~to~apply~an~ICC~profile~but~the~device~color~space~
40313   '#1'~is~unknown.
40314 }
40315 \msg_new:nnnn { color } { ICCBased-unsupported-colorspace }
40316 { ICCBased~color~space~'#1'~uses~an~unsupported~data~color~space. }
40317 {
40318   LaTeX~has~been~asked~to~create~a~ICCBased~colorspace,~but~the~
40319   used~data~colorspace~is~not~supported.~ICC~profiles~used~for~
40320   defining~a~ICCBased~colorspace~should~use~a~Lab,~RGB,~or~
40321   CMYK~data~colorspace.~LaTeX~will~ignore~this~request.
40322 }
```

```

40323 \msg_new:nnnn { color } { ICCBased-requires-file }
40324 { ICCBased-color-space~'#1'~require-an-file. }
40325 {
40326   LaTeX~has~been~asked~to~create~an~ICCBased~color~space,~but~no~\\ \\
40327   \iow_indent:n { file~=<name> }
40328   \\ \\
40329   key~was~given~with~the~correct~information.~LaTeX~will~ignore~this~
40330   request.
40331 }
40332 \msg_new:nnnn { color } { model-already-defined }
40333 { Color~model~'#1'~already~defined. }
40334 {
40335   LaTeX~was~asked~to~define~a~new~color~model~called~'#1',~but~
40336   this~color~model~already~exists.
40337 }
40338 \msg_new:nnnn { color } { out-of-range }
40339 { Input~value~#1~out~of~range~[#2,~#3]. }
40340 {
40341   LaTeX~was~expecting~a~value~in~the~range~[#2,~#3]~as~part~of~a~color,~
40342   but~you~gave~#1.~LaTeX~will~assume~you~meant~the~limit~of~the~range~
40343   and~continue.
40344 }
40345 \msg_new:nnnn { color } { separation-alternative-model }
40346 { Separation~color~space~'#1'~require-an-alternative-model. }
40347 {
40348   LaTeX~has~been~asked~to~create~a~separation~color~space,~
40349   but~no~\\ \\
40350   \iow_indent:n { alternative-model~=<model> }
40351   \\ \\
40352   key~was~given~with~the~correct~information.
40353 }
40354 \msg_new:nnnn { color } { separation-alternative-values }
40355 { Separation~color~space~'#1'~require~values~for~the~alternative~space. }
40356 {
40357   LaTeX~has~been~asked~to~create~a~separation~color~space,~
40358   but~no~\\ \\
40359   \iow_indent:n { alternative-values~=<model> }
40360   \\ \\
40361   key~was~given~with~the~correct~information.
40362 }
40363 \msg_new:nnnn { color } { separation-requires-name }
40364 { Separation~color~space~'#1'~require~a~formal~name. }
40365 {
40366   LaTeX~has~been~asked~to~create~a~separation~color~space,~
40367   but~no~\\ \\
40368   \iow_indent:n { name~=<formal-name> }
40369   \\ \\
40370   key~was~given~with~the~correct~information.
40371 }
40372 \msg_new:nnn { color } { unhandled-model }
40373 {
40374   Unhandled~color~model~in~LaTeX2e~value~"#1":
40375   \\ \\
40376   falling~back~on~grayscale.

```

```

40377 }
40378 \msg_new:nnnn { color } { unknown-color }
40379 { Unknown~color~'#1'. }
40380 {
40381   LaTeX~has~been~asked~to~use~a~color~named~'#1',~
40382   but~this~has~never~been~defined.
40383 }
40384 \msg_new:nnnn { color } { unknown-alternative-model }
40385 { Separation~color~space~'#1'~require~an~valid~alternative~space. }
40386 {
40387   LaTeX~has~been~asked~to~create~a~separation~color~space,~
40388   but~the~model~given~as~\\ \\
40389   \iow_indent:n { alternative-model~==<model> }
40390   \\ \\
40391   is~unknown.
40392 }
40393 \msg_new:nnnn { color } { unknown-export-format }
40394 { Unknown~export~format~'#1'. }
40395 {
40396   LaTeX~has~been~asked~to~export~a~color~in~format~'#1',~
40397   but~this~has~never~been~defined.
40398 }
40399 \msg_new:nnnn { color } { unknown-CIELAB-illuminant }
40400 { Unknown~illuminant~model~'#1'. }
40401 {
40402   LaTeX~has~been~asked~to~create~a~color~space~using~CIELAB~
40403   illuminant~'#1',~but~this~does~not~exist.
40404 }
40405 \msg_new:nnnn { color } { unknown-model }
40406 { Unknown~color~model~'#1'. }
40407 {
40408   LaTeX~has~been~asked~to~use~a~color~model~called~'#1',~
40409   but~this~model~is~not~set~up.
40410 }
40411 \msg_new:nnnn { color } { unknown-model-type }
40412 { Unknown~color~model~type~'#1'. }
40413 {
40414   LaTeX~has~been~asked~to~create~a~new~color~model~called~'#1',~
40415   but~this~type~of~model~was~never~set~up.
40416 }
40417 \prop_gput:Nnn \g_msg_module_name_prop { color } { LaTeX }
40418 \prop_gput:Nnn \g_msg_module_type_prop { color } { }
40419 \msg_new:nnn { color } { show }
40420 {
40421   The~color~#1~
40422   \tl_if_empty:nTF {#2}
40423     { is~undefined. }
40424     { has~the~properties: #2 }
40425 }
40426 </code>

```

Chapter 100

l3graphics implementation

```
40427 <*code>
40428 <@@=graphics>

\l__graphics_tmp_dim Scratch space.
\l__graphics_tmp_ior 40429 \dim_new:N \l__graphics_tmp_dim
\l__graphics_tmp_tl 40430 \ior_new:N \l__graphics_tmp_ior
40431 \tl_new:N \l__graphics_tmp_tl

(End of definition for \l__graphics_tmp_dim, \l__graphics_tmp_ior, and \l__graphics_tmp_tl.)

\s__graphics_stop Internal scan marks.
40432 \scan_new:N \s__graphics_stop

(End of definition for \s__graphics_stop.)
```

100.1 Graphics keys

```
\l__graphics_decodearray_str
\l__graphics_draft_bool
\l__graphics_interpolate_bool
\l__graphics_page_int
\l__graphics_pagebox_tl
\l__graphics_pdf_str
\l__graphics_type_str

40433 \tl_new:N \l__graphics_pagebox_tl
40434 \keys_define:nn { graphics }
40435 {
40436   decodearray .str_set:N =
40437     \l__graphics_decodearray_str ,
40438   draft .bool_set:N =
40439     \l__graphics_draft_bool ,
40440   interpolate .bool_set:N =
40441     \l__graphics_interpolate_bool ,
40442   pagebox .choices:nn =
40443     { art , bleed , crop , media , trim }
40444     {
40445       \tl_set:N \l__graphics_pagebox_tl
40446         { \l_keys_choice_tl box }
40447     } ,
```

```

40448     pagebox .initial:n =
40449         crop ,
40450     page .int_set:N =
40451         \l__graphics_page_int ,
40452     pdf-attr .str_set:N =
40453         \l__graphics_pdf_str ,
40454     type .str_set:N =
40455         \l__graphics_type_str
40456 }

```

(End of definition for `\l__graphics_decodearray_str` and others.)

100.2 Obtaining bounding box data

`\l__graphics_llx_dim` Storage for the return of bounding box.

```

\l__graphics_lly_dim
\l__graphics_urx_dim
\l__graphics_ury_dim
40457 \dim_new:N \l__graphics_llx_dim
40458 \dim_new:N \l__graphics_lly_dim
40459 \dim_new:N \l__graphics_urx_dim
40460 \dim_new:N \l__graphics_ury_dim

```

(End of definition for `\l__graphics_llx_dim` and others.)

`\__graphics_bb_save:n`
`\__graphics_bb_save:e`
`\__graphics_bb_restore:nF`
`\__graphics_bb_restore:eF`

Caching graphic bounding boxes is sensible, and these functions are needed both here and for drive-specific work. So they are made available as documented functions. To save on registers, the “origin” is only saved if it is not at zero.

```

40461 \cs_new_protected:Npn \__graphics_bb_save:n #1
40462 {
40463     \dim_if_exist:cTF { c__graphics_ #1 _urx_dim }
40464     { \msg_error:nnn { graphic } { bb-already-cached } {#1} }
40465     {
40466         \dim_compare:nNnF \l__graphics_llx_dim = { Opt }
40467         { \dim_const:cn { c__graphics_ #1 _llx_dim } { \l__graphics_llx_dim } }
40468         \dim_compare:nNnF \l__graphics_lly_dim = { Opt }
40469         { \dim_const:cn { c__graphics_ #1 _lly_dim } { \l__graphics_lly_dim } }
40470         \dim_const:cn { c__graphics_ #1 _urx_dim } { \l__graphics_urx_dim }
40471         \dim_const:cn { c__graphics_ #1 _ury_dim } { \l__graphics_ury_dim }
40472     }
40473 }
40474 \cs_generate_variant:Nn \__graphics_bb_save:n { e }
40475 \cs_new_protected:Npn \__graphics_bb_restore:nF #1#2
40476 {
40477     \dim_if_exist:cTF { c__graphics_ #1 _urx_dim }
40478     {
40479         \dim_set_eq:Nc \l__graphics_urx_dim { c__graphics_ #1 _urx_dim }
40480         \dim_set_eq:Nc \l__graphics_ury_dim { c__graphics_ #1 _ury_dim }
40481         \dim_if_exist:cTF { c__graphics_ #1 _llx_dim }
40482         { \dim_set_eq:Nc \l__graphics_llx_dim { c__graphics_ #1 _llx_dim } }
40483         { \dim_zero:N \l__graphics_llx_dim }
40484         \dim_if_exist:cTF { c__graphics_ #1 _lly_dim }
40485         { \dim_set_eq:Nc \l__graphics_lly_dim { c__graphics_ #1 _lly_dim } }
40486         { \dim_zero:N \l__graphics_lly_dim }
40487     }
40488     {#2}

```

```

40489 }
40490 \cs_generate_variant:Nn \__graphics_bb_restore:nF { e }

```

(End of definition for __graphics_bb_save:n and __graphics_bb_restore:nF.)

```

\__graphics_extract_bb:n  Extracting the bounding box from an .eps or .bb file is done by reading each line and
\__graphics_read_bb:n     searching for the marker text %%BoundingBox:. The data is read as a string with a
\__graphics_extract_bb_aux:n mapping over the lines: as there is a colon involved, a little bit of work is needed to get
\__graphics_extract_bb_aux:Vn the matching correct. The same approach covers cases where the bounding box has to
\__graphics_extract_bb_auxii:nmn be calculated by extractbb, with just the initial phase different.
\__graphics_extract_bb_auxiii:nmmn
\__graphics_extract_bb_auxiii:Vmmn
\__graphics_extract_bb_auxiv:nmmn
\__graphics_read_bb_auxi:nmmn
\__graphics_read_bb_auxii:Vmmn
\__graphics_read_bb_auxii:w
\__graphics_read_bb_auxiv:w
\__graphics_read_bb_auxv:w

40491 \cs_new_protected:Npn \__graphics_extract_bb:n #1
40492 {
40493   \int_compare:nNnTF \l__graphics_page_int > 0
40494     { \__graphics_extract_bb_auxi:Vn \l__graphics_page_int {#1} }
40495     { \__graphics_extract_bb_auxii:nmn {#1} { } { } }
40496 }
40497 \cs_new_protected:Npn \__graphics_extract_bb_auxi:n #1#2
40498 { \__graphics_extract_bb_auxii:nmn {#2} { :P #1 } { -p~#1~ } }
40499 \cs_generate_variant:Nn \__graphics_extract_bb_auxi:n { Vn }
40500 \cs_new_protected:Npn \__graphics_extract_bb_auxii:nmn #1#2#3
40501 {
40502   \tl_if_empty:NTF \l__graphics_pagebox_tl
40503     { \__graphics_extract_bb_auxiv:nmmn }
40504     { \__graphics_extract_bb_auxiii:Vmmn \l__graphics_pagebox_tl }
40505     {#1} {#2} {#3}
40506 }
40507 \cs_new_protected:Npn \__graphics_extract_bb_auxiii:nmmn #1#2#3#4
40508 { \__graphics_extract_bb_auxiv:nmmn {#2} { : #1 #3 } { #4 -B~#1~ } }
40509 \cs_generate_variant:Nn \__graphics_extract_bb_auxiii:nmmn { V }
40510 \cs_new_protected:Npn \__graphics_extract_bb_auxiv:nmmn #1#2#3
40511 {
40512   \__graphics_read_bb_auxi:nmmn {#1} {#2}
40513   { \ior_shell_open:Nn \l__graphics_tmp_ior { extractbb~#3-0~#1 } }
40514   { pipe-failed }
40515 }
40516 \cs_new_protected:Npn \__graphics_read_bb:n #1
40517 {
40518   \__graphics_read_bb_auxi:nmmn {#1} { }
40519   { \ior_open:Nn \l__graphics_tmp_ior {#1} }
40520   { graphic-not-found }
40521 }
40522 % \end{macrocode}
40523 % Before any real searching, a check to see if there are cached values
40524 % available. The name of each graphic will be unique and so it's sensible to
40525 % store the bounding box data in \TeX{}: this avoids multiple file operations.
40526 % As bounding boxes here start away from the lower-left origin, we need to
40527 % store all four values (contrast with for example the \texttt{pdfmode}
40528 % driver). Here |#2| is a potential page identifier: used to allow caching of
40529 % individual pages in a multi-page document. Caching is applied to the
40530 % upper-right position in all cases, but as the lower-left will often be
40531 % $(0,0)$ it is only cached if required.
40532 % \begin{macrocode}
40533 \cs_new_protected:Npn \__graphics_read_bb_auxi:nmmn #1#2#3#4
40534 {

```

```

40535 \__graphics_bb_restore:nF {#1#2}
40536 { \__graphics_read_bb_auxii:nnnn {#3} {#4} {#1} {#2} }
40537 }
40538 \cs_new_protected:Npe \__graphics_read_bb_auxii:nnnn #1#2#3#4
40539 {
40540 #1
40541 \exp_not:N \ior_if_eof:NTF \exp_not:N \l__graphics_tmp_ior
40542 { \msg_error:nnn { graphics } {#2} {#3} }
40543 {
40544 \ior_str_map_inline:Nn \exp_not:N \l__graphics_tmp_ior
40545 {
40546 \exp_not:N \__graphics_read_bb_auxiii:w
40547 ##1 ~ \c_colon_str \s__graphics_stop
40548 }
40549 \__graphics_bb_save:n {#3#4}
40550 }
40551 \ior_close:N \exp_not:N \l__graphics_tmp_ior
40552 }
40553 \use:e
40554 {
40555 \cs_new_protected:Npn \exp_not:N \__graphics_read_bb_auxiii:w
40556 #1 \c_colon_str #2 \s__graphics_stop
40557 {
40558 \exp_not:N \str_if_eq:nnT
40559 { \c_percent_str \c_percent_str BoundingBox }
40560 {#1}
40561 { \exp_not:N \__graphics_read_bb_auxiv:w #2 ( ) \s__graphics_stop }
40562 }
40563 }

```

If the bounding box is `atend`, just ignore the current one and keep going. Otherwise, we need to allow for tabs and multiple spaces (as the line has been read as a string). The easiest way to deal with that is to scan the tokens: at this stage the line fragment should be just numbers and whitespace. `TeX` will then tidy up for us, with just a leading space to worry about: that is taken out by the `\use:n` here.

```

40564 \cs_new_protected:Npn \__graphics_read_bb_auxiv:w #1 ( #2 ) #3 \s__graphics_stop
40565 {
40566 \str_if_eq:nnF {#2} { atend }
40567 {
40568 \tl_set_rescan:Nne \l__graphics_tmp_tl
40569 {
40570 \char_set_catcode_space:n { 9 }
40571 \char_set_catcode_space:n { 32 }
40572 }
40573 { \use:n #1 }
40574 \exp_after:wN \__graphics_read_bb_auxv:w \l__graphics_tmp_tl \s__graphics_stop
40575 }
40576 }

```

A trailing space was deliberately added earlier so we know that the final data point is terminated by a space.

```

40577 \cs_new_protected:Npn \__graphics_read_bb_auxv:w #1~#2~#3~#4~#5 \s__graphics_stop
40578 {
40579 \dim_set:Nn \l__graphics_llx_dim { #1 bp }

```



```

40580 \dim_set:Nn \l__graphics_lly_dim { #2 bp }
40581 \dim_set:Nn \l__graphics_urx_dim { #3 bp }
40582 \dim_set:Nn \l__graphics_ury_dim { #4 bp }
40583 \ior_map_break:
40584 }

```

(End of definition for __graphics_extract_bb:n and others.)

`\l__graphics_final_name_str` The full name is as you'd expect the name including path and extension. The final
`\l__graphics_full_name_str` name here reflects any conversions carried out by the backend, for example if an `.eps` is
converted to `.pdf`.

```

40585 \str_new:N \l__graphics_final_name_str
40586 \str_new:N \l__graphics_full_name_str

```

(End of definition for \l__graphics_final_name_str and \l__graphics_full_name_str.)

`\l__graphics_tmp_box`

```

40587 \box_new:N \l__graphics_tmp_box

```

(End of definition for \l__graphics_tmp_box.)

`\l__graphics_name_str`, `\l__graphics_ext_str`

```

40588 \str_new:N \l__graphics_dir_str
40589 \str_new:N \l__graphics_name_str
40590 \str_new:N \l__graphics_ext_str

```

(End of definition for \l__graphics_dir_str \l__graphics_name_str \l__graphics_ext_str.)

`\l_graphics_search_path_seq`

```

40591 \seq_new:N \l_graphics_search_path_seq

```

(End of definition for \l_graphics_search_path_seq. This variable is documented on page 340.)

`\l_graphics_search_ext_seq` Used to specify fall-back extensions: actually set on a per-backend basis.

```

40592 \seq_new:N \l_graphics_search_ext_seq

```

(End of definition for \l_graphics_search_ext_seq. This variable is documented on page 340.)

`\l_graphics_ext_type_prop` Mapping between extensions and types for non-standard mappings

```

40593 \prop_new:N \l_graphics_ext_type_prop
40594 \prop_put:Nnn \l_graphics_ext_type_prop { .ps } { eps }

```

(End of definition for \l_graphics_ext_type_prop. This variable is documented on page 340.)

`\g__graphics_record_seq` A list of graphic files used.

```

40595 \seq_new:N \g__graphics_record_seq

```

(End of definition for \g__graphics_record_seq.)

`\graphics_include:nn`

`\graphics_include:nV`

`\__graphics_include_search:n`

`\__graphics_include:`

`\__graphics_include_auxi:n`

`\__graphics_include_auxi:e`

`\__graphics_include_auxii:n`

`\__graphics_include_auxiii:n`

`\__graphics_include_auxiv:n`

Actually including an graphic is relatively straight-forward: most of the work is done by the backend. We only have to deal with making sure the box has no apparent depth. Where the first given name is not found, we search based on extension only if the `<type>` was not given. The one wrinkle is that we may have found a `.tex` file matching the file name stem: that's not what we want, so we have to filter out.

```
40596 \cs_new_protected:Npn \graphics_include:nn #1#2
40597 {
40598   \group_begin:
40599     \keys_set:nn { graphics } {#1}
40600     \seq_set_eq:NN \l_file_search_path_seq \l_graphics_search_path_seq
40601     \file_get_full_name:nNTF {#2} \l_graphics_full_name_str
40602     {
40603       \str_if_eq:eeTF { \l_graphics_full_name_str } { #2 .tex }
40604       { \msg_error:nnn { graphics } { graphic-not-found } {#2} }
40605       { \__graphics_include: }
40606     }
40607     { \msg_error:nnn { graphics } { graphic-not-found } {#2} }
40608   \group_end:
40609 }
40610 \cs_generate_variant:Nn \graphics_include:nn { nV }
40611 \cs_new_protected:Npn \__graphics_include:
40612 {
40613   \str_if_empty:NTF \l__graphics_type_str
40614   {
40615     \file_parse_full_name:VNNN \l__graphics_full_name_str
40616     \l__graphics_dir_str \l__graphics_name_str \l__graphics_ext_str
40617     \__graphics_include_auxi:e
40618     {
40619       \exp_args:Ne \str_tail:n
40620       { \str_casefold:V \l__graphics_ext_str }
40621     }
40622   }
40623   { \__graphics_include_auxi:e { \l__graphics_type_str } }
40624 }
40625 \cs_new_protected:Npn \__graphics_include_auxi:n #1
40626 {
40627   \prop_get:NnNF \l_graphics_ext_type_prop { .#1 } \l_graphics_tmp_tl
40628   { \tl_set:Nn \l__graphics_tmp_tl {#1} }
40629   \exp_args:NV \__graphics_include_auxii:n \l_graphics_tmp_tl
40630 }
40631 \cs_generate_variant:Nn \__graphics_include_auxi:n { e }
40632 \cs_new_protected:Npn \__graphics_include_auxii:n #1
40633 {
40634   \mode_leave_vertical:
40635   \cs_if_exist:cTF { __graphics_backend_include_ #1 :n }
40636   {
40637     \tl_set_eq:NN \l__graphics_final_name_str \l__graphics_full_name_str
40638     \str_set:Ne \l__graphics_full_name_str
40639     { \exp_args:NV __kernel_file_name_quote:n \l__graphics_full_name_str }
40640     \exp_args:NnV \use:c { __graphics_backend_getbb_ #1 :n }
40641     \l__graphics_full_name_str
40642     \seq_gput_right:NV \g__graphics_record_seq \l__graphics_final_name_str
40643     \clist_if_exist:NT \@filelist
```

```

40644         { \exp_args:NV \@addtofilelist \l__graphics_final_name_str }
40645         \bool_if:NTF \l__graphics_draft_bool
40646         { \__graphics_include_auxiii:n }
40647         { \__graphics_include_auxiv:n }
40648         {#1}
40649     }
40650     { \msg_error:nnn { graphics } { unsupported-graphic-type } {#1} }
40651 }
40652 \cs_new_protected:Npn \__graphics_include_auxiii:n #1
40653 {
40654     \hbox_to_wd:nn { \l__graphics_urx_dim - \l__graphics_llx_dim }
40655     {
40656         \tex_vrule:D
40657         \tex_hss:D
40658         \vbox_to_ht:nn
40659         { \l__graphics_ury_dim - \l__graphics_lly_dim }
40660         {
40661             \tex_hrule:D width
40662             \dim_eval:n { \l__graphics_urx_dim - \l__graphics_llx_dim }
40663             \tex_vss:D
40664             \hbox_to_wd:nn
40665             { \l__graphics_urx_dim - \l__graphics_llx_dim }
40666             {
40667                 \ttfamily
40668                 \tex_hss:D \l__graphics_full_name_str \tex_hss:D
40669             }
40670             \tex_vss:D
40671             \tex_hrule:D
40672         }
40673         \tex_hss:D
40674         \tex_vrule:D
40675     }
40676 }
40677 \cs_new_protected:Npn \__graphics_include_auxiv:n #1
40678 {
40679     \hbox_set:Nn \l__graphics_tmp_box
40680     {
40681         \exp_args:NnV \use:c { __graphics_backend_include_ #1 :n }
40682         \l__graphics_full_name_str
40683     }
40684     \box_set_dp:Nn \l__graphics_tmp_box { Opt }
40685     \box_set_ht:Nn \l__graphics_tmp_box
40686     { \l__graphics_ury_dim - \l__graphics_lly_dim }
40687     \box_set_wd:Nn \l__graphics_tmp_box
40688     { \l__graphics_urx_dim - \l__graphics_llx_dim }
40689     \box_use_drop:N \l__graphics_tmp_box
40690 }

```

(End of definition for \graphics_include:nn and others. This function is documented on page 340.)

```

\graphics_show_list: A function to list all graphic files used.
\graphics_log_list: 40691 \cs_new_protected:Npn \graphics_show_list: { \__graphics_list:N \msg_show:nneeee }
\__graphics_list:N 40692 \cs_new_protected:Npn \graphics_log_list: { \__graphics_list:N \msg_log:nneeee }
\__graphics_list_aux:n 40693 \cs_new_protected:Npn \__graphics_list:N #1

```

```

40694 {
40695   \seq_remove_duplicates:N \g__graphics_record_seq
40696   #1 { kernel } { file-list }
40697   { \seq_map_function:NN \g__graphics_record_seq \__graphics_list_aux:n }
40698   { } { } { }
40699 }
40700 \cs_new:Npn \__graphics_list_aux:n #1 { \iow_newline: #1 }

```

(End of definition for `\graphics_show_list:` and others. These functions are documented on page 341.)

100.3 Utility functions

`\graphics_get_full_name:nN`
`\graphics_get_full_name:nNTF`
`\__graphics_get_full_name:n`

As well as searching by path, etc., there is a need here to check that we do not trip over `foo.bar` if `.bar` is not a known extension for the current backend.

```

40701 \cs_new_protected:Npn \graphics_get_full_name:nN #1#2
40702 {
40703   \graphics_get_full_name:nNF {#1} #2
40704   { \tl_set:Nn #2 { \q_no_value } }
40705 }
40706 \prg_new_protected_conditional:Npnn \graphics_get_full_name:nN #1#2
40707 { T , F , TF }
40708 {
40709   \group_begin:
40710   \seq_set_eq:NN \l_file_search_path_seq \l_graphics_search_path_seq
40711   \file_get_full_name:nNTF {#1} \l__graphics_full_name_str
40712   {
40713     \str_if_eq:eeTF { \l__graphics_full_name_str } { #1 .tex }
40714     { \__graphics_get_full_name:n {#1} }
40715     {
40716       \file_parse_full_name:VNNN \l__graphics_full_name_str
40717       \l__graphics_dir_str \l__graphics_name_str \l__graphics_ext_str
40718       \seq_map_inline:Nn \l_graphics_search_ext_seq
40719       {
40720         \str_if_eq:nVT {##1} \l__graphics_ext_str
40721         { \seq_map_break:n { \use_none:nn } }
40722       }
40723       \__graphics_get_full_name:n {#1}
40724     }
40725   }
40726   { \__graphics_get_full_name:n {#1} }
40727   \exp_args:NNNV \group_end:
40728   \tl_set:Nn #2 \l__graphics_full_name_str
40729   \tl_if_empty:NNTF #2
40730   { \prg_return_false: }
40731   { \prg_return_true: }
40732 }
40733 \cs_new_protected:Npn \__graphics_get_full_name:n #1
40734 {
40735   \str_clear:N \l__graphics_full_name_str
40736   \seq_map_inline:Nn \l_graphics_search_ext_seq
40737   {
40738     \file_get_full_name:nNT { #1 ##1 } \l__graphics_full_name_str

```

```

40739         { \seq_map_break:n { \use_none:nn } }
40740     }
40741     \use:n
40742     { \str_clear:N \l__graphics_full_name_str }
40743 }

```

(End of definition for `\graphics_get_full_name:nN`, `\graphics_get_full_name:nNTF`, and `\__graphics_get_full_name:n`. These functions are documented on page 340.)

`\graphics_get_pagecount:nN` A generic function to read the number of pages in a graphic file. This is used by all of the backend where there is not a dedicated primitive. The plan is essentially the same as reading the bounding box. To avoid multiple calls, the value is cached either here or in the backend.

```

40744 \cs_new_protected:Npn \graphics_get_pagecount:nN #1#2
40745 {
40746     \group_begin:
40747     \seq_set_eq:NN \l_file_search_path_seq \l_graphics_search_path_seq
40748     \file_get_full_name:nNTF {#1} \l__graphics_full_name_str
40749     {
40750         \int_if_exist:cF { c__graphics_ \l__graphics_full_name_str _pages_int }
40751         {
40752             \exp_args:NV \__graphics_backend_get_pagecount:n
40753             \l__graphics_full_name_str
40754         }
40755         \tl_set:Nv #2 { c__graphics_ \l__graphics_full_name_str _pages_int }
40756     }
40757     {
40758         \tl_set:Nn #2 { 0 }
40759         \msg_error:nnn { graphics } { graphic-not-found } {#1}
40760     }
40761     \exp_args:NNNV \group_end:
40762     \tl_set:Nn #2 #2
40763 }
40764 \cs_new_protected:Npe \__graphics_get_pagecount:n #1
40765 {
40766     \exp_not:N \ior_shell_open:Nn \exp_not:N \l__graphics_tmp_ior
40767     { extractbb--0~#1 }
40768     \exp_not:N \ior_if_eof:NTF \exp_not:N \l__graphics_tmp_ior
40769     { \msg_error:nnn { graphics } { pipe-failed } }
40770     {
40771         \ior_str_map_inline:Nn \exp_not:N \l__graphics_tmp_ior
40772         {
40773             \exp_not:N \__graphics_get_pagecount:nw {#1}
40774             #1 ~ \c_colon_str \c_colon_str \s__graphics_stop
40775         }
40776         \exp_not:N \int_if_exist:cF { c__graphics_ #1 _pages_int }
40777         { \int_const:cn { c__graphics_ #1 _pages_int } { 1 } }
40778     }
40779     \ior_close:N \exp_not:N \l__graphics_tmp_ior
40780 }
40781 \use:e
40782 {
40783     \cs_new_protected:Npn \exp_not:N \__graphics_get_pagecount:nw
40784     #1#2 \c_colon_str #3 \c_colon_str #4 \s__graphics_stop

```

```

40785     {
40786       \exp_not:N \str_if_eq:nnT
40787       { \c_percent_str \c_percent_str Pages }
40788       {#2}
40789       {
40790         \int_const:cn { c__graphics_ #1 _pages_int } {#3}
40791         \exp_not:N \ior_map_break:
40792       }
40793     }
40794   }

```

(End of definition for `\graphics_get_pagecount:nN`, `\__graphics_get_pagecount:n`, and `\__graphics_get_pagecount:nw`. This function is documented on page [340](#).)

100.4 Messages

```

40795 \msg_new:nnnn { graphics } { graphic-not-found }
40796 { Image~file~'#1'~not~found. }
40797 {
40798   LaTeX~tried~to~open~graphic~file~'#1',~
40799   but~the~file~could~not~be~read.
40800 }
40801 \msg_new:nnnn { graphics } { pipe-failed }
40802 { Cannot~run~piped~system~commands. }
40803 {
40804   LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
40805   Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
40806 }
40807 \msg_new:nnnn { graphics } { unsupported-graphic-type }
40808 { Image~type~'#1'~not~supported~by~current~driver. }
40809 {
40810   LaTeX~was~asked~to~include~an~graphic~of~type~'#1',~
40811   but~this~is~not~supported~by~the~current~driver~(production~route).
40812 }
40813 </code>

```

Chapter 101

l3opacity implementation

```
40814 \*code)
40815 \<@opacity>
    Transitional support.
40816 \cs_if_exist:NT \@expl@finalise@setup@@
40817 {
40818     \tl_gput_right:Nn \@expl@finalise@setup@@
40819     { \declare@file@substitution { l3opacity.sty } { null.tex } }
40820 }
```

\l__opacity_tmp_fp Temporary storage.

```
40821 \fp_new:N \l__opacity_tmp_fp
```

(End of definition for \l__opacity_tmp_fp.)

```
\opacity_select:n Thin wrapper with error checking. Opacity is passed to backend functions as a bounded,
\opacity_fill:n    evaluated decimal number.
\opacity_stroke:n
\opacity_begin:n
\opacity_fill_begin:n
\opacity_stroke_begin:n
\opacity_end:
\opacity_fill_end:
\opacity_stroke_end:
\__opacity_select:nNn
\__opacity_select:nN
40822 \cs_new_protected:Npn \opacity_select:n #1
40823 {
40824     \__opacity_select:nNn {#1} \__opacity_backend_select:n
40825     { \group_insert_after:N \__opacity_backend_reset: }
40826 }
40827 \cs_new_protected:Npn \opacity_fill:n #1
40828 {
40829     \__opacity_select:nNn {#1} \__opacity_backend_fill:n
40830     { \group_insert_after:N \__opacity_backend_reset_fill: }
40831 }
40832 \cs_new_protected:Npn \opacity_stroke:n #1
40833 {
40834     \__opacity_select:nNn {#1} \__opacity_backend_stroke:n
40835     { \group_insert_after:N \__opacity_backend_reset_stroke: }
40836 }
40837 \cs_new_protected:Npn \opacity_begin:n #1
40838 { \__opacity_select:nN {#1} \__opacity_backend_select:n }
40839 \cs_new_protected:Npn \opacity_fill_begin:n #1
40840 { \__opacity_select:nN {#1} \__opacity_backend_fill:n }
40841 \cs_new_protected:Npn \opacity_stroke_begin:n #1
40842 { \__opacity_select:nN {#1} \__opacity_backend_stroke:n }
40843 \cs_new_protected:Npn \opacity_end:
```

```

40844 { \_opacity_backend_reset: }
40845 \cs_new_protected:Npn \opacity_fill_end:
40846 { \_opacity_backend_reset_fill: }
40847 \cs_new_protected:Npn \opacity_stroke_end:
40848 { \_opacity_backend_reset_stroke: }
40849 \cs_new_protected:Npn \_opacity_select:nNn #1#2#3
40850 {
40851   \fp_set:Nn \l__opacity_tmp_fp {#1}
40852   \bool_lazy_or:nnTF
40853     { \fp_compare_p:nNn \l__opacity_tmp_fp < \c_zero_fp }
40854     { \fp_compare_p:nNn \l__opacity_tmp_fp > \c_one_fp }
40855     { \msg_error:nnn { opacity } { out-of-range } {#1} }
40856     {
40857       \exp_args:Ne #2 { \fp_use:N \l__opacity_tmp_fp }
40858       #3
40859     }
40860 }
40861 \cs_new_protected:Npn \_opacity_select:nN #1#2
40862 { \_opacity_select:nNn {#1} #2 { } }
40863 \msg_new:nnnn { opacity } { out-of-range }
40864 { Opacity~value~out~of~range. }
40865 {
40866   LaTeX~was~asked~to~set~opacity~of~#1,~but~only~values~in~the~range~
40867   0~to~1~are~supported.
40868 }

```

(End of definition for `\opacity_select:n` and others. These functions are documented on page [342](#).)

```

40869 \code

```


Chapter 102

l3pdf implementation

```
40870 <@@=pdf>
40871 <*code>

\s__pdf_stop Internal scan marks.
40872 \scan_new:N \s__pdf_stop

(End of definition for \s__pdf_stop.)

\g__pdf_init_bool A boolean so we have some chance of avoiding setting things we are not allowed to. As
we are potentially early in the format, we have to work a bit harder than ideal.
40873 \bool_new:N \g__pdf_init_bool
40874 \bool_lazy_and:nnT
40875 { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
40876 { \tl_if_exist_p:N \@expl@finalise@setup@@ }
40877 {
40878   \tl_gput_right:Nn \@expl@finalise@setup@@
40879   {
40880     \tl_gput_right:Nn \@kernel@after@begindocument
40881     { \bool_gset_true:N \g__pdf_init_bool }
40882   }
40883 }

(End of definition for \g__pdf_init_bool.)
```

102.1 Compression

```
\pdf_uncompress: Simple to do.
40884 \cs_new_protected:Npn \pdf_uncompress:
40885 {
40886   \bool_if:NF \g__pdf_init_bool
40887   {
40888     \__pdf_backend_compresslevel:n { 0 }
40889     \__pdf_backend_compress_objects:n { \c_false_bool }
40890   }
40891 }

(End of definition for \pdf_uncompress:. This function is documented on page 345.)
```

102.2 Objects

`\g__pdf_backend_object_int` For returning object numbers.

```
40892 \int_new:N \g__pdf_backend_object_int
```

(End of definition for `\g__pdf_backend_object_int`.)

Simple to do: all objects create a constant `int` so it is not a backend-specific name.

```

\pdf_object_new:n
\pdf_object_write:nnn
\pdf_object_write:nne
\pdf_object_write:nnx
\pdf_object_ref:n
\__kernel_pdf_object_id:n
40893 \cs_new_protected:Npn \pdf_object_new:n #1
40894 {
40895   \__pdf_backend_object_new:
40896   \__pdf_object_record:nN {#1} \g__pdf_backend_object_int
40897 }
40898 \cs_new_protected:Npn \pdf_object_write:nnn #1#2#3
40899 {
40900   \exp_args:Ne \__pdf_backend_object_write:nnn
40901   { \__pdf_object_retrieve:n {#1} } {#2} {#3}
40902   \bool_gset_true:N \g__pdf_init_bool
40903 }
40904 \cs_generate_variant:Nn \pdf_object_write:nnn { nne , nnx }
40905 \cs_new:Npn \pdf_object_ref:n #1
40906 {
40907   \exp_args:Ne \__pdf_backend_object_ref:n
40908   { \__pdf_object_retrieve:n {#1} }
40909 }
40910 \cs_new:Npn \__kernel_pdf_object_id:n #1
40911 {
40912   \exp_args:Ne \__pdf_backend_object_id:n
40913   { \__pdf_object_retrieve:n {#1} }
40914 }
40915 \code

```

(End of definition for `\pdf_object_new:n` and others. These functions are documented on page 343.)

`\__pdf_object_record:nN`
`\__pdf_object_retrieve:n`
`ltx.pdf.object_id` Object mappings are tracked in Lua for LuaTeX as this makes retrieving them much easier; as a result, there is a split in approaches. In Lua we store values in a table indexed by name. The Lua function here is set up to deal with both named and indexed objects: fits the Lua idiom well.

```

40916 (*lua)
40917
40918 local scan_int = token.scan_int
40919 local scan_string = token.scan_string
40920 local cprint = tex.cprint
40921
40922 local __pdf_objects_named = {}
40923 local __pdf_objects_indexed = {}
40924
40925 luacmd('__pdf_object_record:nN', function()
40926   local name = scan_string()
40927   local n = scan_int()
40928   __pdf_objects_named[name] = n
40929 end, 'protected', 'global')
40930
40931 local function object_id(name, index)

```

```

40932   if index then
40933     return __pdf_objects_indexed[name][index] or 0
40934   else
40935     return __pdf_objects_named[name] or 0
40936   end
40937 end
40938
40939 luacmd('__pdf_object_retrieve:n', function()
40940   local name = scan_string()
40941   return cprint(12,toststring(object_id(name)))
40942 end,'global')
40943
40944 ltx.pdf = ltx.pdf or {}
40945 ltx.pdf.object_id = object_id
40946
40947 </lua>

```

Whereas in T_EX we use integer constants.

```

40948 <*code>
40949 \sys_if_engine luatex:F
40950 {
40951   \cs_new_protected:Npn \__pdf_object_record:nN #1#2
40952   {
40953     \int_const:cn
40954     { c__pdf_object_ #1 _int } {#2}
40955   }
40956   \cs_new:Npn \__pdf_object_retrieve:n #1
40957   {
40958     \int_if_exist:cTF { c__pdf_object_ #1 _int }
40959     {
40960       \int_use:c
40961       { c__pdf_object_ #1 _int }
40962     }
40963     { 0 }
40964   }
40965 }

```

(End of definition for `\__pdf_object_record:nN`, `\__pdf_object_retrieve:n`, and `ltx.pdf.object_id`. This function is documented on page ??.)

```

\pdf_object_if_exist_p:n
\pdf_object_if_exist:nTF
40966 \prg_new_conditional:Npnn \pdf_object_if_exist:n #1 { p , T , F , TF }
40967 {
40968   \int_compare:nNnTF { \__pdf_object_retrieve:n {#1} } = 0
40969   \prg_return_false:
40970   \prg_return_true:
40971 }

```

(End of definition for `\pdf_object_if_exist:nTF`. This function is documented on page 343.)

```

\pdf_object_new_indexed:nn
\pdf_object_write_indexed:nnnn
\pdf_object_write_indexed:nnne
\pdf_object_ref_indexed:nn
\__kernel_pdf_object_id_indexed:nn
40972 \cs_new_protected:Npn \pdf_object_new_indexed:nn #1#2
40973 {

```

Again we split between the common code and the macro- or Lua-based implementation. To make life easier for the Lua route, all of the potential expressions are expanded to braced numbers.

```

40974 \__pdf_backend_object_new:
40975 \__pdf_object_record:neN {#1}
40976 { \int_eval:n {#2} } \g__pdf_backend_object_int
40977 }
40978 \cs_new_protected:Npn \pdf_object_write_indexed:nnnn #1#2#3#4
40979 {
40980 \exp_args:Ne \__pdf_backend_object_write:nnn
40981 { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } } {#3} {#4}
40982 \bool_gset_true:N \g__pdf_init_bool
40983 }
40984 \cs_generate_variant:Nn \pdf_object_write_indexed:nnnn { nnne }
40985 \cs_new:Npn \pdf_object_ref_indexed:nn #1#2
40986 {
40987 \exp_args:Ne \__pdf_backend_object_ref:n
40988 { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
40989 }
40990 \cs_new:Npn \__kernel_pdf_object_id_indexed:nn #1#2
40991 {
40992 \exp_args:Ne \__pdf_backend_object_id:n
40993 { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
40994 }
40995 \code

```

(End of definition for `\pdf_object_new_indexed:nn` and others. These functions are documented on page 344.)

```

\__pdf_object_record:nnN
\__pdf_object_record:neN
\__pdf_object_retrieve:nn
\__pdf_object_record:NnN
\__pdf_object_retrieve:Nn

```

Again we split for Lua: the same idea as above but with nested tables. As we've arranged above that the TeX code passes a braced number, we can use `tonumber(scan_string())` rather than `scan_int()` for the index.

```

40996 \lua
40997
40998 luacmd('__pdf_object_record:nnN', function()
40999 local name = scan_string()
41000 local index = tonumber(scan_string())
41001 local n = scan_int()
41002 __pdf_objects_indexed[name] = __pdf_objects_indexed[name] or {}
41003 __pdf_objects_indexed[name][index] = n
41004 end, 'protected', 'global')
41005
41006 luacmd('__pdf_object_retrieve:nn', function()
41007 local name = scan_string()
41008 local index = tonumber(scan_string())
41009 return cprint(12, tostring(object_id(name, index)))
41010 end, 'global')
41011
41012 \lua

```

The non-Lua approach is to divide the range into blocks, and store in integer arrays that can simulate dynamic assignment.

```

41013 \code
41014 \sys_if_engine luatex:F
41015 {
41016 \cs_new_protected:Npn \__pdf_object_record:nnN #1#2#3
41017 {
41018 \use:e

```

```

41019         {
41020             \__pdf_object_record:NnN
41021             \__pdf_object_index_split:nn {#1} {#2}
41022             \exp_not:N #3
41023         }
41024     }
41025     \cs_new_protected:Npn \__pdf_object_record:NnN #1#2#3
41026     {
41027         \intarray_if_exist:NF #1
41028         { \intarray_new:Nn #1 \c__pdf_object_block_size_int }
41029         \intarray_gset:Nnn #1 {#2} #3
41030     }
41031     \cs_new:Npn \__pdf_object_retrieve:nn #1#2
41032     {
41033         \use:e
41034         {
41035             \exp_not:N \__pdf_object_retrieve:Nn
41036             \__pdf_object_index_split:nn {#1} {#2}
41037         }
41038     }
41039     \cs_new:Npn \__pdf_object_retrieve:Nn #1#2
41040     { \intarray_item:Nn #1 {#2} }

```

As we want blocks to start from one, and within the block for the top value to be “in” the block, we do a little bit of manipulation. By shifting down by one, we keep the values “in” the block, then we adjust the block/index number to get back on track.

```

41041     \cs_new:Npn \__pdf_object_index_split:nn #1#2
41042     {
41043         \exp_not:c
41044         {
41045             g__pdf_object_ #1 _
41046             \int_eval:n
41047             {
41048                 \int_div_truncate:nn { #2 - 1 }
41049                 \c__pdf_object_block_size_int + 1
41050             }
41051             _intarray
41052         }
41053         {
41054             \int_eval:n
41055             { \int_mod:nn { #2 - 1 } \c__pdf_object_block_size_int + 1 }
41056         }
41057     }

```

(End of definition for __pdf_object_record:nnN and others.)

\c__pdf_object_block_size_int Sets the block size used for managing indexed objects.

```

41058     \int_const:Nn \c__pdf_object_block_size_int { 10000 }
41059 }

```

(End of definition for \c__pdf_object_block_size_int.)

__pdf_object_record:neN Common variants.

```

\__pdf_object_retrieve:ne 41060 \cs_generate_variant:Nn \__pdf_object_record:nnN { ne }
41061 \cs_generate_variant:Nn \__pdf_object_retrieve:nn { ne }

```

(End of definition for _pdf_object_record:neN and _pdf_object_retrieve:ne.)

\pdf_object_unnamed_write:nn No tracking needed here.

```
\pdf_object_unnamed_write:ne 41062 \cs_new_protected:Npn \pdf_object_unnamed_write:nn #1#2
\pdf_object_unnamed_write:nx 41063 {
41064   \exp_args:Ne \_pdf_backend_object_now:nn {#1} {#2}
41065   \bool_gset_true:N \g__pdf_init_bool
41066 }
41067 \cs_generate_variant:Nn \pdf_object_unnamed_write:nn { ne , nx }
```

(End of definition for \pdf_object_unnamed_write:nn. This function is documented on page 344.)

\pdf_object_ref_last: A one-step wrapper for consistency.

```
41068 \cs_new:Npn \pdf_object_ref_last: { \_pdf_backend_object_last: }
```

(End of definition for \pdf_object_ref_last:. This function is documented on page 345.)

\pdf_pageobject_ref:n

```
41069 \cs_new:Npn \pdf_pageobject_ref:n #1
41070 { \exp_args:Ne \_pdf_backend_pageobject_ref:n {#1} }
```

(End of definition for \pdf_pageobject_ref:n. This function is documented on page 345.)

102.3 Version

\pdf_version_compare_p:Nn To compare version, we need to split the given value then deal with both major and
\pdf_version_compare:NnTF minor version

```
__pdf_version_compare_=:w 41071 \prg_new_conditional:Npnn \pdf_version_compare:Nn #1#2 { p , T , F , TF }
__pdf_version_compare_<:w 41072 { \use:c { __pdf_version_compare_ #1 :w } #2 . . \s__pdf_stop }
__pdf_version_compare_>:w 41073 \cs_new:cpn { __pdf_version_compare_=:w } #1 . #2 . #3 \s__pdf_stop
41074 {
41075   \bool_lazy_and:nnTF
41076   { \int_compare_p:nNn \_pdf_backend_version_major: = {#1} }
41077   { \int_compare_p:nNn \_pdf_backend_version_minor: = {#2} }
41078   { \prg_return_true: }
41079   { \prg_return_false: }
41080 }
41081 \cs_new:cpn { __pdf_version_compare_<:w } #1 . #2 . #3 \s__pdf_stop
41082 {
41083   \bool_lazy_or:nnTF
41084   { \int_compare_p:nNn \_pdf_backend_version_major: < {#1} }
41085   {
41086     \bool_lazy_and:p:nn
41087     { \int_compare_p:nNn \_pdf_backend_version_major: = {#1} }
41088     { \int_compare_p:nNn \_pdf_backend_version_minor: < {#2} }
41089   }
41090   { \prg_return_true: }
41091   { \prg_return_false: }
41092 }
41093 \cs_new:cpn { __pdf_version_compare_>:w } #1 . #2 . #3 \s__pdf_stop
41094 {
41095   \bool_lazy_or:nnTF
41096   { \int_compare_p:nNn \_pdf_backend_version_major: > {#1} }
```

```

41097     {
41098         \bool_lazy_and_p:nn
41099         { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
41100         { \int_compare_p:nNn \__pdf_backend_version_minor: > {#2} }
41101     }
41102     { \prg_return_true: }
41103     { \prg_return_false: }
41104 }

```

(End of definition for `\pdf_version_compare:NnTF` and others. This function is documented on page 345.)

```

\pdf_version_gset:n Split the version and set.
\pdf_version_min_gset:n
\__pdf_version_gset:w
41105 \cs_new_protected:Npn \pdf_version_gset:n #1
41106 { \__pdf_version_gset:w #1 . . \s__pdf_stop }
41107 \cs_new_protected:Npn \pdf_version_min_gset:n #1
41108 {
41109     \pdf_version_compare:NnT < {#1}
41110     { \__pdf_version_gset:w #1 . . \s__pdf_stop }
41111 }
41112 \cs_new_protected:Npn \__pdf_version_gset:w #1 . #2 . #3\s__pdf_stop
41113 {
41114     \bool_if:NF \g__pdf_init_bool
41115     {
41116         \__pdf_backend_version_major_gset:n {#1}
41117         \__pdf_backend_version_minor_gset:n {#2}
41118     }
41119 }

```

(End of definition for `\pdf_version_gset:n`, `\pdf_version_min_gset:n`, and `\__pdf_version_gset:w`. These functions are documented on page 345.)

```

\pdf_version: Wrappers.
\pdf_version_major:
\pdf_version_minor:
41120 \cs_new:Npn \pdf_version:
41121 { \__pdf_backend_version_major: . \__pdf_backend_version_minor: }
41122 \cs_new:Npn \pdf_version_major: { \__pdf_backend_version_major: }
41123 \cs_new:Npn \pdf_version_minor: { \__pdf_backend_version_minor: }

```

(End of definition for `\pdf_version:`, `\pdf_version_major:`, and `\pdf_version_minor:`. These functions are documented on page 345.)

102.4 Page size

```

\pdf_pagesize_gset:nn
41124 \cs_new_protected:Npn \pdf_pagesize_gset:nn #1#2
41125 { \__pdf_backend_pagesize_gset:nn {#1} {#2} }

```

(End of definition for `\pdf_pagesize_gset:nn`. This function is documented on page 345.)

102.5 Destinations

`\pdf_destination:nn`

```
41126 \cs_new_protected:Npn \pdf_destination:nn #1#2
41127 { \__pdf_backend_destination:nn {#1} {#2} }
```

(End of definition for `\pdf_destination:nn`. This function is documented on page 346.)

`\pdf_destination:nnnn`

```
41128 \cs_new_protected:Npn \pdf_destination:nnnn #1#2#3#4
41129 {
41130   \hbox_to_zero:n
41131   { \__pdf_backend_destination:nnnn {#1} {#2} {#3} {#4} }
41132 }
```

(End of definition for `\pdf_destination:nnnn`. This function is documented on page 346.)

102.6 PDF Page size (media box)

Everything here is delayed to the start of the document so that the backend will definitely be loaded.

```
41133 \cs_if_exist:NT \@kernel@before@begindocument
41134 {
41135   \tl_gput_right:Nn \@kernel@before@begindocument
41136   {
41137     \bool_lazy_all:nT
41138     {
41139       { \cs_if_exist_p:N \stockheight }
41140       { \cs_if_exist_p:N \stockwidth }
41141       { \cs_if_exist_p:N \IfPDFManagementActiveTF }
41142       { \IfPDFManagementActiveTF { \c_true_bool } { \c_false_bool } }
41143       { \int_compare_p:nNn \tex_mag:D = { 1000 } }
41144     }
41145     {
41146       \bool_lazy_and:nnTF
41147       { \dim_compare_p:nNn \stockheight > { Opt } }
41148       { \dim_compare_p:nNn \stockwidth > { Opt } }
41149       {
41150         \__pdf_backend_pagesize_gset:nn
41151         \stockwidth \stockheight
41152       }
41153       {
41154         \bool_lazy_or:nnF
41155         { \dim_compare_p:nNn \stockheight < { Opt } }
41156         { \dim_compare_p:nNn \stockwidth < { Opt } }
41157         {
41158           \bool_lazy_and:nnT
41159           { \dim_compare_p:nNn \paperheight > { Opt } }
41160           { \dim_compare_p:nNn \paperwidth > { Opt } }
41161           {
41162             \__pdf_backend_pagesize_gset:nn
41163             \paperwidth \paperheight
41164           }
41165         }
41166       }
41167     }
41168   }
41169 }
```



```
41165     }  
41166   }  
41167 }  
41168 }  
41169 }  
41170 </code>
```

Chapter 103

l3benchmark implementation

Our working unit is the scaled second, namely 2^{-16} seconds.

```
41171 (*code)
```

103.1 Benchmarking code

```
41172 (@@=benchmark)
```

```
\g_benchmark_duration_target_fp
```

The benchmark is constrained to take roughly (from half to twice) `\g_benchmark_duration_target_fp` seconds, unless one iteration of the code takes longer.

```
41173 \fp_new:N \g_benchmark_duration_target_fp
```

```
41174 \fp_gset:Nn \g_benchmark_duration_target_fp { 1 }
```

(End of definition for `\g_benchmark_duration_target_fp`. This variable is documented on page 348.)

103.1.1 Raw measurement

```
\g__benchmark_nesting_int  
  \__benchmark_raw:nN  
  \__benchmark_raw_aux:N  
  \__benchmark_raw_end:N
```

Store in the given integer variable the time it took to perform a given piece of code, in scaled seconds. We call `\sys_timer:` as close before and after the code as possible. We store the intermediate result in a new integer when `\__benchmark_raw:nN` is nested.

```
41175 \int_new:N \g__benchmark_nesting_int
```

```
41176 \cs_new_protected:Npn \__benchmark_raw:nN #1
```

```
41177 {
```

```
41178   \int_gincr:N \g__benchmark_nesting_int
```

```
41179   \exp_args:Nc \__benchmark_raw_aux:N
```

```
41180     { g__benchmark_ \int_use:N \g__benchmark_nesting_int _int }
```

```
41181   \__benchmark_raw_aux:
```

```
41182     #1
```

```
41183   \__benchmark_raw_end:N
```

```
41184 }
```

```
41185 \cs_new_protected:Npn \__benchmark_raw_aux:N #1
```

```
41186 {
```

```
41187   \int_gzero_new:N #1
```

```
41188   \cs_gset_protected:Npn \__benchmark_raw_aux: { \int_gset:Nn #1 { \sys_timer: } }
```

```
41189 }
```

```
41190 \cs_new_protected:Npn \__benchmark_raw_end:N #1
```

```
41191 {
```

```

41192 \int_gset:Nn #1
41193 {
41194   \sys_timer: -
41195   \int_use:c { g__benchmark_ \int_use:N \g__benchmark_nesting_int _int }
41196 }
41197 \int_gdecr:N \g__benchmark_nesting_int
41198 }

```

(End of definition for `\g__benchmark_nesting_int` and others.)

```

\__benchmark_raw_replicate:nnN
\__benchmark_tmp:w

```

Here, we wish to measure the time it takes for the piece of code #2 to be run #1 times, and store the result in the integer #3.

If the number of copies required is large, defining `\__benchmark_tmp:w` would exhaust T_EX's main memory. In that case, we replicate #1/5000 times the given code before passing it to the main call to `\__benchmark_tmp:w`. Of course the division rounds to an integer, so that step introduces a relative error of order at most 5000/500000, less than many other sources of variability.

We subtract the time for another call to `\__benchmark_tmp:w`, with the same arguments (to capture the time it takes to read the argument) but empty expansion.

```

41199 \cs_new_eq:NN \__benchmark_tmp:w ?
41200 \cs_new_protected:Npn \__benchmark_raw_replicate:nnN #1
41201 {
41202   \int_compare:nNnTF {#1} > { 500000 }
41203   { \__benchmark_raw_replicate_large:nnN {#1} }
41204   { \__benchmark_raw_replicate_small:nnN {#1} }
41205 }
41206 \cs_new_protected:Npn \__benchmark_raw_replicate_large:nnN #1#2
41207 {
41208   \cs_set:Npe \__benchmark_tmp:w ##1 { \prg_replicate:nn { 5000 } {##1} }
41209   \exp_args:Nno \__benchmark_raw_replicate:nnN { #1 / 5000 }
41210   { \__benchmark_tmp:w {#2} }
41211 }
41212 \cs_new_protected:Npn \__benchmark_raw_replicate_small:nnN #1#2
41213 {
41214   \cs_set:Npe \__benchmark_tmp:w ##1##2 { \prg_replicate:nn {#1} {##1} }
41215   \__benchmark_raw:nN { \__benchmark_tmp:w {#2} { } } \g__benchmark_time_int
41216   \exp_args:No \__benchmark_raw_replicate_aux:nnN
41217   { \int_use:N \g__benchmark_time_int } {#2}
41218 }
41219 \cs_new_protected:Npn \__benchmark_raw_replicate_aux:nnN #1#2#3
41220 {
41221   \__benchmark_raw:nN { \__benchmark_tmp:w { } {#2} } \g__benchmark_time_int
41222   \int_gset:Nn #3 { #1 - \g__benchmark_time_int }
41223   \cs_set_eq:NN \__benchmark_tmp:w \prg_do_nothing:
41224 }

```

(End of definition for `\__benchmark_raw_replicate:nnN` and `\__benchmark_tmp:w`.)

103.1.2 Main benchmarking

```

\g_benchmark_time_fp
\g_benchmark_ops_fp

```

Functions such as `\benchmark:n` store the measured time in `\g_benchmark_time_fp` (in seconds) and the estimated number of operations in `\g_benchmark_ops_fp`.

```

41225 \fp_new:N \g_benchmark_time_fp
41226 \fp_new:N \g_benchmark_ops_fp

```

(End of definition for `\g_benchmark_time_fp` and `\g_benchmark_ops_fp`. These variables are documented on page 348.)

`\g__benchmark_duration_int` A conversion of `\g_benchmark_duration_target_fp` seconds into scaled seconds.

```
41227 \int_new:N \g__benchmark_duration_int
```

(End of definition for `\g__benchmark_duration_int`.)

`\g__benchmark_time_int` These variables hold the time for running a piece of code, as an integer in scaled seconds.

```
\g__benchmark_time_a_int 41228 \int_new:N \g__benchmark_time_int
\g__benchmark_time_b_int 41229 \int_new:N \g__benchmark_time_a_int
\g__benchmark_time_c_int 41230 \int_new:N \g__benchmark_time_b_int
\g__benchmark_time_d_int 41231 \int_new:N \g__benchmark_time_c_int
                          41232 \int_new:N \g__benchmark_time_d_int
```

(End of definition for `\g__benchmark_time_int` and others.)

`\g__benchmark_repeat_int` Holds the number of times that the piece of code was repeated when timing.

```
41233 \int_new:N \g__benchmark_repeat_int
```

(End of definition for `\g__benchmark_repeat_int`.)

`\g__benchmark_code_tl` Holds the piece of code to repeat.

```
41234 \tl_new:N \g__benchmark_code_tl
```

(End of definition for `\g__benchmark_code_tl`.)

`\benchmark_once:n` Convert the raw time from scaled seconds to seconds, and convert to a number of operations. It is important to measure the elementary operation before running the user code because both measurements use the same temporary variables.

`\benchmark_once_silent:n`

```
41235 \cs_new_protected:Npn \benchmark_once:n #1
41236 {
41237   \benchmark_once_silent:n {#1}
41238   \__benchmark_display:
41239 }
41240 \cs_new_protected:Npn \benchmark_once_silent:n #1
41241 {
41242   \__benchmark_measure_op:
41243   \__benchmark_raw:nN {#1} \g__benchmark_time_int
41244   \fp_gset:Nn \g_benchmark_time_fp { \g__benchmark_time_int / 65536 }
41245   \fp_gset:Nn \g_benchmark_ops_fp { \g_benchmark_time_fp / \g__benchmark_one_op_fp }
41246 }
```

(End of definition for `\benchmark_once:n` and `\benchmark_once_silent:n`. These functions are documented on page 348.)

`\benchmark:n` After setting up some variables the work is done by `\__benchmark_aux:.`

```
41247 \cs_new_protected:Npn \benchmark:n #1
41248 {
41249   \benchmark_silent:n {#1}
41250   \__benchmark_display:
41251 }
41252 \cs_new_protected:Npn \benchmark_silent:n #1
41253 {
41254   \__benchmark_measure_op:
```

```

41255 \tl_gset:Nn \g__benchmark_code_tl {#1}
41256 \__benchmark_aux:
41257 \fp_gset:Nn \g_benchmark_ops_fp { \g_benchmark_time_fp / \g__benchmark_one_op_fp }
41258 }

```

(End of definition for \benchmark:n. This function is documented on page 348.)

__benchmark_aux: The main timing function. First time the user code once. If that took more than half the allotted time (\g__benchmark_duration_int) we're done. If that took much less, repeatedly quadruple the number of copies until it takes a reasonable amount of time. Once we reach a reasonable time (or we risk an overflow), compute a number of times that can fit in one quarter of the allotted time and measure that four times. To save time we reuse the result of the first pass if \g__benchmark_repeat_int is one. Once we have four results, find the smallest, divided by 65536 and by the number of repetitions, and display that.

```

41259 \cs_new_protected:Npn \__benchmark_aux:
41260 {
41261   \int_gset:Nn \g__benchmark_repeat_int { 1 }
41262   \__benchmark_raw:nN { \g__benchmark_code_tl } \g__benchmark_time_int
41263   \int_compare:nNnF \g__benchmark_time_int < { \g__benchmark_duration_int / 2 }
41264     { \prg_break: }
41265   \bool_until_do:nn
41266     {
41267       \int_compare_p:nNn \g__benchmark_time_int > { \g__benchmark_duration_int / 32 }
41268       || \int_compare_p:nNn \g__benchmark_repeat_int > { \c_max_int / 4 }
41269     }
41270     {
41271       \int_gset:Nn \g__benchmark_repeat_int { 4 * \g__benchmark_repeat_int }
41272       \__benchmark_run:N \g__benchmark_time_int
41273     }
41274   \int_gset:Nn \g__benchmark_repeat_int
41275     {
41276       \fp_to_int:n
41277       {
41278         max ( 1 , min ( \c_max_int ,
41279           \g__benchmark_duration_int * \g__benchmark_repeat_int /
41280           \int_eval:n { 4 * \g__benchmark_time_int } ) )
41281       }
41282     }
41283   \int_compare:nNnTF \g__benchmark_repeat_int = 1
41284     { \int_gset_eq:NN \g__benchmark_time_a_int \g__benchmark_time_int }
41285     { \__benchmark_run:N \g__benchmark_time_a_int }
41286   \__benchmark_run:N \g__benchmark_time_b_int
41287   \__benchmark_run:N \g__benchmark_time_c_int
41288   \__benchmark_run:N \g__benchmark_time_d_int
41289   \int_gset:Nn \g__benchmark_time_int
41290     {
41291       \int_min:nn
41292       { \int_min:nn \g__benchmark_time_a_int \g__benchmark_time_b_int }
41293       { \int_min:nn \g__benchmark_time_c_int \g__benchmark_time_d_int }
41294     }
41295   \prg_break_point:
41296   \int_compare:nNnT \g__benchmark_time_int < 3 { \int_gzero:N \g__benchmark_time_int }
41297   \fp_gset:Nn \g_benchmark_time_fp

```

```

41298         { \g__benchmark_time_int / \g__benchmark_repeat_int / 65536 }
41299     }

```

```

41300 \cs_new_protected:Npn \__benchmark_run:N

```

```

41301     { \exp_args:NNo \__benchmark_raw_replicate:nnN \g__benchmark_repeat_int { \g__benchmark_c

```

(End of definition for __benchmark_aux:.)

103.1.3 Display

\g__benchmark_one_op_fp Time for one operation.

```

41302 \fp_new:N \g__benchmark_one_op_fp

```

(End of definition for \g__benchmark_one_op_fp.)

__benchmark_measure_op: Measure one arbitrary single operation (which we put in \g__benchmark_code_tl). This uses a common auxiliary __benchmark_aux: with the main benchmark function.

```

41303 \cs_new_protected:Npn \__benchmark_measure_op:

```

```

41304     {
41305         \int_gset:Nn \g__benchmark_duration_int
41306         { \fp_to_int:n { 65536 * \g_benchmark_duration_target_fp } / 4 }
41307         \tl_gset:Nn \g__benchmark_code_tl

```

```

41308         { \int_gadd:Nn \g__benchmark_duration_int { 0 } }

```

```

41309         \__benchmark_aux:

```

```

41310         \fp_gset:Nn \g__benchmark_one_op_fp { max(\g_benchmark_time_fp, 1e-8) }

```

```

41311         \int_gset:Nn \g__benchmark_duration_int

```

```

41312         { \fp_to_int:n { 65536 * \g_benchmark_duration_target_fp } }

```

```

41313     }

```

(End of definition for __benchmark_measure_op:.)

__benchmark_fp_to_tl:N Similar to \fp_to_tl:N but rounds to 3 significant digits and uses scientific notation starting from 1e3.

```

41314 \cs_new:Npn \__benchmark_fp_to_tl:N #1

```

```

41315     {
41316         \fp_compare:nTF { abs(#1) < 1000 }
41317         { \fp_to_tl:n { round(#1, 2 - logb(#1)) } }
41318         {

```

```

41319             \exp_args:Nf \__benchmark_fp_to_tl_aux:nN

```

```

41320             { \fp_to_int:n { logb(#1) } } #1

```

```

41321         }

```

```

41322     }

```

```

41323 \cs_new:Npn \__benchmark_fp_to_tl_aux:nN #1#2

```

```

41324     { \fp_to_tl:n { round(#2 * 1e-#1, 2) } e#1 }

```

(End of definition for __benchmark_fp_to_tl:N and __benchmark_fp_to_tl_aux:nN.)

__benchmark_display: Function to display the time that was measured and the estimated number of operations.

```

41325 \cs_new_protected:Npn \__benchmark_display:

```

```

41326     {

```

```

41327         \iow_term:e

```

```

41328         {

```

```

41329             \__benchmark_fp_to_tl:N \g_benchmark_time_fp \c_space_tl seconds \c_space_tl

```

```

41330             ( \__benchmark_fp_to_tl:N \g_benchmark_ops_fp \c_space_tl ops)

```

```

41331         }

```

```

41332     }

```

(End of definition for __benchmark_display:.)

103.2 Benchmark tic toc

```

\g__benchmark_tictoc_int
\g__benchmark_tictoc_seq
\l__benchmark_tictoc_pop_tl
41333 \int_new:N \g__benchmark_tictoc_int
41334 \seq_new:N \g__benchmark_tictoc_seq
41335 \tl_new:N \l__benchmark_tictoc_pop_tl

(End of definition for \g__benchmark_tictoc_int, \g__benchmark_tictoc_seq, and \l__benchmark_tictoc_pop_tl.)

\__benchmark_tictoc_prefix: We include the package name in analogy with continuation lines of error/warning messages.
41336 \cs_new:Npn \__benchmark_tictoc_prefix:
41337 {
41338   (l3benchmark) \c_space_tl
41339   + \prg_replicate:nn { \g__benchmark_tictoc_int } { -+ } \c_space_tl
41340 }

(End of definition for \__benchmark_tictoc_prefix:.)

\benchmark_tic:
41341 \cs_new_protected:Npn \benchmark_tic:
41342 {
41343   \iow_term:e { \__benchmark_tictoc_prefix: TIC }
41344   \exp_args:NNf \seq_gput_right:Nn \g__benchmark_tictoc_seq { \sys_timer: }
41345   \int_gincr:N \g__benchmark_tictoc_int
41346 }

(End of definition for \benchmark_tic:. This function is documented on page 349.)

\benchmark_toc:
\__benchmark_toc:
41347 \cs_new:Npn \benchmark_toc:
41348 {
41349   \seq_gpop_right:NNTF \g__benchmark_tictoc_seq \l__benchmark_tictoc_pop_tl
41350   { \__benchmark_toc: }
41351   { \msg_error:nn { benchmark } { toc-first } }
41352 }
41353 \cs_new_protected:Npn \__benchmark_toc:
41354 {
41355   \int_gdecr:N \g__benchmark_tictoc_int
41356   \fp_gset:Nn \g__benchmark_time_fp
41357   { ( \sys_timer: - \l__benchmark_tictoc_pop_tl ) / 65536 }
41358   \iow_term:e
41359   {
41360     \__benchmark_tictoc_prefix:
41361     TOC: \c_space_tl
41362     \__benchmark_fp_to_tl:N \g__benchmark_time_fp \c_space_tl s
41363   }
41364 }
41365 \msg_new:nnn { benchmark } { toc-first }
41366 {
41367   \token_to_str:N \benchmark_toc: \c_space_tl without~
41368   \token_to_str:N \benchmark_tic: \c_space_tl !
41369 }

```

(End of definition for \benchmark_toc: and _benchmark_toc:. This function is documented on page 349.)

41370 `</code>`

Chapter 104

l3deprecation implementation

```
41371 <{*code}>
41372 <@@=deprecation>
```

104.1 Patching definitions to deprecate

```
  \_kernel_patch_deprecation:nnNNpn {<date>} {<replacement>} {<definition>}
  {<function>} {<parameters>} {<code>}
```

defines the *<function>* to produce an error and run its *<code>*.

We make `\debug_on:n {deprecation}` turn the *<function>* into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behavior was without `\debug_on:n {deprecation}`.

In the explanations below, *<definition>* *<function>* *<parameters>* {*<code>*} or assignments that only differ in the scope of the *<definition>* will be called “the standard definition”.

```
\_kernel_patch_deprecation:nnNNpn (The parameter text is grabbed using #5#.) The arguments of \_kernel_deprecation_
\_deprecation_patch_aux:nnNNnn code:nn are run upon \debug_on:n {deprecation} and \debug_off:n {deprecation},
\_deprecation_warn_once:nnNnn respectively. In both scenarios we the <function> may be \outer so we undefine it with
\_deprecation_patch_aux:Nn \tex_let:D before redefining it, with \_kernel_deprecation_error:Nnn or with some
\_deprecation_just_error:nnNN code added shortly.

41373 \cs_new_protected:Npn \_kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
41374 { \_deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
41375 \cs_new_protected:Npn \_deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
41376 {
41377   \_kernel_deprecation_code:nn
41378   {
41379     \tex_let:D #4 \scan_stop:
41380     \_kernel_deprecation_error:Nnn #4 {#2} {#1}
41381   }
41382   { \tex_let:D #4 \scan_stop: }
41383   \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
41384   { \_deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
41385   { \_deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
41386 }
```

In case we want a warning, the `\function` is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the `\function` should have, with arguments, and call that definition. The `e`-type expansion and `\exp_not:n` avoid needing to double the `#`, which we could not do anyways. We then deal with the code for `\debug_off:n {deprecation}`: presumably someone doing that does not need the warning so we simply do the standard definition.

```

41387 \cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
41388 {
41389   \cs_gset_protected:Npe #3
41390   {
41391     \__kernel_if_debug:TF
41392     {
41393       \exp_not:N \msg_warning:nneee
41394       { deprecation } { deprecated-command }
41395       {#1}
41396       { \token_to_str:N #3 }
41397       { \tl_to_str:n {#2} }
41398     }
41399     { }
41400     \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
41401     \exp_not:N #3
41402   }
41403   \__kernel_deprecation_code:nn { }
41404   { \cs_set_protected:Npn #3 #4 {#5} }
41405 }

```

In case we want neither warning nor error, the `\function` is given its standard definition. Here `#1` is `\cs_new:Npn` or `\cs_new_protected:Npn` and `#2` is `\function` `\parameters` `{\code}`, so `#1#2` performs the assignment. For `\debug_off:n {deprecation}` we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```

41406 \cs_new_protected:Npn \__deprecation_patch_aux:Nn #1#2
41407 {
41408   #1 #2
41409   \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
41410   { \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
41411   { \__kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
41412 }

```

(End of definition for `\__kernel_patch_deprecation:nnNNpn` and others.)

`\__kernel_deprecation_error:Nnn` The `\outer` definition here ensures the command cannot appear in an argument.

```

41413 \cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
41414 {
41415   \tex_protected:D \tex_outer:D \tex_edef:D #1
41416   {
41417     \exp_not:N \msg_expandable_error:nnnnn
41418     { deprecation } { deprecated-command }
41419     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
41420     \exp_not:N \msg_error:nneee
41421     { deprecation } { deprecated-command }
41422     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
41423   }
41424 }

```

(End of definition for `\_kernel_deprecation_error:Nnn`.)

```

41425 \msg_new:nnn { deprecation } { deprecated-command }
41426 {
41427   \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
41428   #2-deprecated-on~#1.
41429 }

```

104.2 Deprecated **l3**basics functions

41430 `<@@=cs>`

`\cs_argument_spec:N` For the present, do not deprecate fully as L^AT_EX 2_ε will need to catch up: one for Fall 2022.

```

41431 %\_kernel_patch_deprecation:nnNNpn { 2022-06-24 } { \cs_parameter_spec:N }
41432 \cs_new:Npn \cs_argument_spec:N { \cs_parameter_spec:N }

```

(End of definition for `\cs_argument_spec:N`.)

104.3 Deprecated **l3**file functions

41433 `<@@=file>`

`\iow_shipout_x:Nn` Previously described as x-type, but the hash behavior is really e-type. Currently not “live” as we need to have a transition.

```

\iow_shipout_x:Nx
\iow_shipout_x:cn
\iow_shipout_x:cx
41434 %\_kernel_patch_deprecation:nnNNpn { 2023-10-10 } { \iow_shipout_e:Nn }
41435 \cs_new_protected:Npn \iow_shipout_x:Nn { \iow_shipout_e:Nn }
41436 \cs_generate_variant:Nn \iow_shipout_x:Nn { Nx , c, cx }

```

(End of definition for `\iow_shipout_x:Nn`.)

104.4 Deprecated **l3**keys functions

41437 `<@@=keys>`

```

.str_set_x:N
.str_set_x:c
41438 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:N } #1
41439 { \__keys_variable_set:NnnN #1 { str } { } x }
.str_gset_x:N
41440 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:c } #1
41441 { \__keys_variable_set:cnnN {#1} { str } { } x }
.str_gset_x:c
41442 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:N } #1
41443 { \__keys_variable_set:NnnN #1 { str } { g } x }
41444 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:c } #1
41445 { \__keys_variable_set:cnnN {#1} { str } { g } x }

```

(End of definition for `.str_set_x:N` and `.str_gset_x:N`.)

```

.tl_set_x:N
.tl_set_x:c
41446 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
41447 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_gset_x:N
41448 \cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
41449 { \__keys_variable_set:cnnN {#1} { tl } { } x }
.tl_gset_x:c
41450 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1

```

```

41451 { \_keys_variable_set:NnnN #1 { t1 } { g } x }
41452 \cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
41453 { \_keys_variable_set:cnnN {#1} { t1 } { g } x }

```

(End of definition for .tl_set_x:N and .tl_gset_x:N.)

```

\keys_set_filter:nnnN We need a transition here so for the present this is commented out: only needed for
\keys_set_filter:nnVN latex-lab code so this should not last for too long.
\keys_set_filter:nnvN 41454 %\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnn }
\keys_set_filter:nnoN 41455 \cs_new_protected:Npn \keys_set_filter:nnn { \keys_set_exclude_groups:nnn }
\keys_set_filter:nnnnN 41456 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
\keys_set_filter:nnVnN 41457 %\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnN }
\keys_set_filter:nnvnN 41458 \cs_new_protected:Npn \keys_set_filter:nnnN { \keys_set_exclude_groups:nnnN }
\keys_set_filter:nnonN 41459 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
\keys_set_filter:nnn 41460 %\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnnN }
\keys_set_filter:nnV 41461 \cs_new_protected:Npn \keys_set_filter:nnnnN { \keys_set_exclude_groups:nnnnN }
\keys_set_filter:nnv 41462 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
\keys_set_filter:nno (End of definition for \keys_set_filter:nnnN, \keys_set_filter:nnnnN, and \keys_set_filter:nnn.)

```

104.5 Deprecated l3msg functions

```

41463 <@@=msg>
\msg_gset:nnnn
\msg_gset:nnn 41464 \__kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnnn }
41465 \cs_new_protected:Npn \msg_gset:nnnn { \msg_set:nnnn }
41466 \__kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnn }
41467 \cs_new_protected:Npn \msg_gset:nnn { \msg_set:nnn }
(End of definition for \msg_gset:nnnn and \msg_gset:nnn.)

```

104.6 Deprecated l3pdf functions

```

41468 <@@=pdf>
\g__pdf_object_prop For tracking objects.
41469 \prop_new:N \g__pdf_object_prop
(End of definition for \g__pdf_object_prop.)

\pdf_object_new:nn
\pdf_object_write:nn 41470 \__kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_new:n] }
\pdf_object_write:nx 41471 \cs_new_protected:Npn \pdf_object_new:nn #1#2
41472 {
41473   \prop_gput:Nnn \g__pdf_object_prop {#1} {#2}
41474   \pdf_object_new:n {#1}
41475 }
41476 \__kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_write:n] }
41477 \cs_new_protected:Npn \pdf_object_write:nn #1#2
41478 {
41479   \exp_args:Nee \__pdf_backend_object_write:nnn
41480   { \__pdf_object_retrieve:n {#1} }
41481   { \prop_item:Nn \g__pdf_object_prop {#1} } {#2}

```

```

41482     \bool_gset_true:N \g__pdf_init_bool
41483   }
41484   \cs_generate_variant:Nn \pdf_object_write:nn { nx }

(End of definition for \pdf_object_new:nn and \pdf_object_write:nn.)

```

104.7 Deprecated l3prg functions

```

41485 <@@=cs>

\bool_case_true:n
\bool_case_true:nTF
41486   \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:n }
41487   \cs_new:Npn \bool_case_true:n { \bool_case:n }
41488   \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nT }
41489   \cs_new:Npn \bool_case_true:nT { \bool_case:nT }
41490   \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nF }
41491   \cs_new:Npn \bool_case_true:nF { \bool_case:nF }
41492   \__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nTF }
41493   \cs_new:Npn \bool_case_true:nTF { \bool_case:nTF }

(End of definition for \bool_case_true:nTF.)

```

104.8 Deprecated l3regex functions

```

41494 <@@=regex>

\regex_match:nnTF
\regex_match:nVTF
\regex_match:NnTF
\regex_match:NVTF
41495   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:nnT }
41496   \cs_new_protected:Npn \regex_match:nnT { \regex_if_match:nnT }
41497   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:nnF }
41498   \cs_new_protected:Npn \regex_match:nnF { \regex_if_match:nnF }
41499   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:nnTF }
41500   \cs_new_protected:Npn \regex_match:nnTF { \regex_if_match:nnTF }
41501   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:NnT }
41502   \cs_new_protected:Npn \regex_match:NnT { \regex_if_match:NnT }
41503   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:NnF }
41504   \cs_new_protected:Npn \regex_match:NnF { \regex_if_match:NnF }
41505   \%__kernel_patch_deprecation:nnNNpn { 2025-05-14 } { \regex_if_match:NnTF }
41506   \cs_new_protected:Npn \regex_match:NnTF { \regex_if_match:NnTF }
41507   \prg_generate_conditional_variant:Nnn \regex_match:nn
41508     { nV } { T , F , TF }
41509   \prg_generate_conditional_variant:Nnn \regex_match:Nn
41510     { NV } { T , F , TF }

(End of definition for \regex_match:nnTF and \regex_match:NnTF.)

```

104.9 Deprecated l3str functions

```

41511 <@@=str>

\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case:f
\str_fold_case:n
\str_fold_case:V
41512   \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
41513   \cs_new:Npn \str_lower_case:n { \str_lowercase:n }

```

```

41514 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:f }
41515 \cs_new:Npn \str_lower_case:f { \str_lowercase:f }
41516 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
41517 \cs_new:Npn \str_upper_case:n { \str_uppercase:n }
41518 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:f }
41519 \cs_new:Npn \str_upper_case:f { \str_uppercase:f }
41520 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
41521 \cs_new:Npn \str_fold_case:n { \str_casefold:n }
41522 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:V }
41523 \cs_new:Npn \str_fold_case:V { \str_casefold:V }

```

(End of definition for `\str_lower_case:n`, `\str_upper_case:n`, and `\str_fold_case:n`.)

```

\str_foldcase:n
\str_foldcase:V

```

```

41524 \__kernel_patch_deprecation:nnNNpn { 2020-10-17 } { \str_casefold:n }
41525 \cs_new:Npn \str_foldcase:n { \str_casefold:n }
41526 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:V }
41527 \cs_new:Npn \str_foldcase:V { \str_casefold:V }

```

(End of definition for `\str_foldcase:n`.)

```
\str_declare_eight_bit_encoding:nnn
```

This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accommodate all of Unicode.

```

41528 \__kernel_patch_deprecation:nnNNpn { 2020-08-20 } { }
41529 \cs_new_protected:Npn \str_declare_eight_bit_encoding:nnn #1
41530 { \__str_declare_eight_bit_encoding:nnnn {#1} { 1114112 } }

```

(End of definition for `\str_declare_eight_bit_encoding:nnn`.)

104.10 Deprecated l3seq functions

```
41531 (@@=seq)
```

```

\seq_indexed_map_inline:Nn
\seq_indexed_map_function:NN

```

```

41532 \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_inline:Nn }
41533 \cs_new_protected:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
41534 \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_function:NN }
41535 \cs_new:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }

```

(End of definition for `\seq_indexed_map_inline:Nn` and `\seq_indexed_map_function:NN`.)

```
\seq_mapthread_function:NNN
```

```

41536 \__kernel_patch_deprecation:nnNNpn { 2023-05-10 } { \seq_map_pairwise_function:NNN }
41537 \cs_new:Npn \seq_mapthread_function:NNN { \seq_map_pairwise_function:NNN }

```

(End of definition for `\seq_mapthread_function:NNN`.)

```

\seq_set_map_x:NNn
\seq_gset_map_x:NNn

```

```

41538 \__kernel_patch_deprecation:nnNNpn { 2023-10-26 } { \seq_set_map_e:NNn }
41539 \cs_new_protected:Npn \seq_set_map_x:NNn { \seq_set_map_e:NNn }
41540 \__kernel_patch_deprecation:nnNNpn { 2023-10-26 } { \seq_gset_map_e:NNn }
41541 \cs_new_protected:Npn \seq_gset_map_x:NNn { \seq_gset_map_e:NNn }

```

(End of definition for `\seq_set_map_x:NNn` and `\seq_gset_map_x:NNn`.)

104.11 Deprecated l3sys functions

41542 $\langle @@=\text{sys} \rangle$

`\sys_finalise:`

41543 `__kernel_patch_deprecation:nnNNpn { 2025-05-25 } { \sys_finalize: }`

41544 `\cs_new_protected:Npn \sys_finalise: { \sys_finalize: }`

(End of definition for \sys_finalise:.)

`\sys_load_deprecation:`

41545 `__kernel_patch_deprecation:nnNNpn { 2021-01-11 } { (no-longer-required) }`

41546 `\cs_new_protected:Npn \sys_load_deprecation: { }`

(End of definition for \sys_load_deprecation:.)

`\sys_if_timer_exist_p:`

`\sys_if_timer_exist:TF`

41547 `__kernel_patch_deprecation:nnNNpn { 2025-03-26 } { (no-longer-required) }`

41548 `\cs_new:Npn \sys_if_timer_exist:T #1 {#1}`

41549 `__kernel_patch_deprecation:nnNNpn { 2025-03-26 } { (no-longer-required) }`

41550 `\cs_new:Npn \sys_if_timer_exist:F #1 { }`

41551 `__kernel_patch_deprecation:nnNNpn { 2025-03-26 } { (no-longer-required) }`

41552 `\cs_new:Npn \sys_if_timer_exist:TF #1#2 {#1}`

41553 `__kernel_patch_deprecation:nnNNpn { 2025-03-26 } { (no-longer-required) }`

41554 `\cs_new:Npn \sys_if_timer_exist_p: { \c_true_bool }`

(End of definition for \sys_if_timer_exist:TF.)

104.12 Deprecated l3text functions

41555 $\langle @@=\text{text} \rangle$

`\text_titlecase:n`

`\text_titlecase:nn`

41556 `__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:n }`

41557 `\cs_new:Npn \text_titlecase:n #1`

41558 `{ \text_titlecase_first:n { \text_lowercase:n {#1} } }`

41559 `__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:nn }`

41560 `\cs_new:Npn \text_titlecase:nn #1#2`

41561 `{ \text_titlecase_first:nn {#1} { \text_lowercase:n {#2} } }`

(End of definition for \text_titlecase:n and \text_titlecase:nn.)

104.13 Deprecated l3tl functions

41562 $\langle @@=\text{tl} \rangle$

`\tl_lower_case:n`

`\tl_lower_case:nn`

`\tl_upper_case:n`

`\tl_upper_case:nn`

`\tl_mixed_case:n`

`\tl_mixed_case:nn`

41563 `__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }`

41564 `\cs_new:Npn \tl_lower_case:n #1`

41565 `{ \text_lowercase:n {#1} }`

41566 `__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:nn }`

41567 `\cs_new:Npn \tl_lower_case:nn #1#2`

41568 `{ \text_lowercase:nn {#1} {#2} }`

41569 `__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }`

```

41570 \cs_new:Npn \tl_upper_case:n #1
41571   { \text_uppercase:n {#1} }
41572 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:nn }
41573 \cs_new:Npn \tl_upper_case:nn #1#2
41574   { \text_uppercase:nn {#1} {#2} }
41575 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:n }
41576 \cs_new:Npn \tl_mixed_case:n #1
41577   { \text_titlecase_first:n { \text_lowercase:n {#1} } }
41578 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:nn }
41579 \cs_new:Npn \tl_mixed_case:nn #1#2
41580   { \text_titlecase_first:nn {#1} { \text_lowercase:n {#2} } }

```

(End of definition for \tl_lower_case:n and others.)

```

\tl_case:Nn
\tl_case:cn
\tl_case:NnTF
\tl_case:cnTF
41581 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:Nn }
41582 \cs_new:Npn \tl_case:Nn { \token_case_meaning:Nn }
41583 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnT }
41584 \cs_new:Npn \tl_case:NnT { \token_case_meaning:NnT }
41585 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnF }
41586 \cs_new:Npn \tl_case:NnF { \token_case_meaning:NnF }
41587 \__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnTF }
41588 \cs_new:Npn \tl_case:NnTF { \token_case_meaning:NnTF }
41589 \cs_generate_variant:Nn \tl_case:Nn { c }
41590 \prg_generate_conditional_variant:Nnn \tl_case:Nn
41591   { c } { T , F , TF }

```

(End of definition for \tl_case:NnTF.)

```

\tl_build_clear:N
\tl_build_gclear:N
41592 \__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_begin:N }
41593 \cs_new_protected:Npn \tl_build_clear:N { \tl_build_begin:N }
41594 \__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_gbegin:N }
41595 \cs_new_protected:Npn \tl_build_gclear:N { \tl_build_gbegin:N }

```

(End of definition for \tl_build_clear:N and \tl_build_gclear:N.)

```

\tl_build_get:NN
41596 \__kernel_patch_deprecation:nnNNpn { 2023-10-25 } { \tl_build_get_intermediate:NN }
41597 \cs_new_protected:Npn \tl_build_get:NN { \tl_build_get_intermediate:NN }

```

(End of definition for \tl_build_get:NN.)

104.14 Deprecated l3token functions

```

41598 <@@=char>
\char_to_utfviii_bytes:n
41599 \__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { [ \codepoint_generate:nn ] }
41600 \cs_new:Npn \char_to_utfviii_bytes:n { \__kernel_codepoint_to_bytes:n }

```

(End of definition for \char_to_utfviii_bytes:n.)

\char_to_nfd:N

\char_to_nfd:n

```
41601 \__kernel_patch_deprecation:nmNnpn { 2022-10-09 } { \codepoint_to_nfd:n }
41602 \cs_new:Npn \char_to_nfd:N #1 { \codepoint_to_nfd:n {‘#1} }
41603 \__kernel_patch_deprecation:nmNnpn { 2022-10-09 } { \codepoint_to_nfd:n }
41604 \cs_new:Npn \char_to_nfd:n { \codepoint_to_nfd:n }
```

(End of definition for \char_to_nfd:N and \char_to_nfd:n.)

\char_lower_case:N

\char_upper_case:N

\char_mixed_case:Nn

\char_fold_case:N

\char_str_lower_case:N

\char_str_upper_case:N

\char_str_mixed_case:N

\char_str_fold_case:N

```
41605 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \text_lowercase:n }
41606 \cs_new:Npn \char_lower_case:N { \text_lowercase:n }
41607 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \text_uppercase:n }
41608 \cs_new:Npn \char_upper_case:N { \text_uppercase:n }
41609 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \text_titlecase_first:n }
41610 \cs_new:Npn \char_mixed_case:N { \text_titlecase_first:n }
41611 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \str_casefold:n }
41612 \cs_new:Npn \char_fold_case:N { \str_casefold:n }
41613 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \str_lowercase:n }
41614 \cs_new:Npn \char_str_lower_case:N { \str_lowercase:n }
41615 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \str_uppercase:n }
41616 \cs_new:Npn \char_str_upper_case:N { \str_uppercase:n }
41617 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \str_titlecase:n }
41618 \cs_new:Npn \char_str_mixed_case:N { \str_titlecase:n }
41619 \__kernel_patch_deprecation:nmNnpn { 2020-01-03 } { \str_casefold:n }
41620 \cs_new:Npn \char_str_fold_case:N { \str_casefold:n }
```

(End of definition for \char_lower_case:N and others.)

\char_lowercase:N

\char_titlecase:N

\char_uppercase:N

\char_foldcase:N

\char_str_lowercase:N

\char_str_titlecase:N

\char_str_uppercase:N

\char_str_foldcase:N

```
41621 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \text_lowercase:n }
41622 \cs_new:Npn \char_lowercase:N { \text_lowercase:n }
41623 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \text_uppercase:n }
41624 \cs_new:Npn \char_uppercase:N { \text_uppercase:n }
41625 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \text_titlecase_first:n }
41626 \cs_new:Npn \char_titlecase:N { \text_titlecase_first:n }
41627 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \str_casefold:n }
41628 \cs_new:Npn \char_foldcase:N { \str_casefold:n }
41629 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \str_lowercase:n }
41630 \cs_new:Npn \char_str_lowercase:N { \str_lowercase:n }
41631 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 }
41632 { \tl_to_str:e { \text_titlecase_first:n } }
41633 \cs_new:Npn \char_str_titlecase:N #1
41634 { \tl_to_str:e { \text_titlecase_first:n {#1} } }
41635 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \str_uppercase:n }
41636 \cs_new:Npn \char_str_uppercase:N { \str_uppercase:n }
41637 \__kernel_patch_deprecation:nmNnpn { 2022-10-17 } { \str_casefold:n }
41638 \cs_new:Npn \char_str_foldcase:N { \str_casefold:n }
```

(End of definition for \char_lowercase:N and others.)

\peek_catcode_ignore_spaces:NTF

A little extra fun here to deal with the expansion.

\peek_catcode_remove_ignore_spaces:NTF

```
41639 \tl_map_inline:nn
```

\peek_charcode_ignore_spaces:NTF

```
41640 {
```

\peek_charcode_remove_ignore_spaces:NTF

```
41641 { catcode } { catcode_remove }
```

\peek_meaning_ignore_spaces:NTF

\peek_meaning_remove_ignore_spaces:NTF

```

41642     { charcode } { charcode_remove }
41643     { meaning } { meaning_remove }
41644 }
41645 {
41646     \use:e
41647     {
41648         \_kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
41649         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NTF } ##1##2##3
41650         {
41651             \peek_remove_spaces:n
41652             { \exp_not:c { peek_ #1 :NTF } ##1 {##2} {##3} }
41653         }
41654         \_kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
41655         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NT } ##1##2
41656         {
41657             \peek_remove_spaces:n
41658             { \exp_not:c { peek_ #1 :NT } ##1 {##2} }
41659         }
41660         \_kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
41661         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NF } ##1##2
41662         {
41663             \peek_remove_spaces:n
41664             { \exp_not:c { peek_ #1 :NF } ##1 {##2} }
41665         }
41666     }
41667 }

```

(End of definition for \peek_catcode_ignore_spaces:NTF and others.)

104.15 Deprecated l3prop functions

```

\prop_put_if_new:Nnn
\prop_put_if_new:NVn
\prop_put_if_new:NnV
\prop_put_if_new:cnn
\prop_put_if_new:cVn
\prop_put_if_new:cnV
\prop_gput_if_new:Nnn
\prop_gput_if_new:NVn
\prop_gput_if_new:NnV
\prop_gput_if_new:cnn
\prop_gput_if_new:cVn
\prop_gput_if_new:cnV

```

```

41668 %\_kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_put_if_not_in:Nnn }
41669 \cs_new_protected:Npn \prop_put_if_new:Nnn { \prop_put_if_not_in:Nnn }
41670 %\_kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_gput_if_not_in:Nnn }
41671 \cs_new_protected:Npn \prop_gput_if_new:Nnn { \prop_gput_if_not_in:Nnn }
41672 \cs_generate_variant:Nn \prop_put_if_new:Nnn
41673 { NnV , NV , c , cnV , cV }
41674 \cs_generate_variant:Nn \prop_gput_if_new:Nnn
41675 { NnV , NV , c , cnV , cV }

```

(End of definition for \prop_put_if_new:Nnn and \prop_gput_if_new:Nnn.)

```

41676 </code>

```

Chapter 105

l3debug implementation

Internal kernel functions that are only defined here are listed in `l3kernel-functions`, see [45.1](#).

```
41677 <*def>
41678 <@@=debug>
      Standard file identification.
41679 \ProvidesExplFile{l3debug.def}{2026-03-16}{L3 Debugging support}
```

`\s__debug_stop` Internal scan marks.

```
41680 \scan_new:N \s__debug_stop
```

(End of definition for \s__debug_stop.)

`\__debug_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
41681 \cs_new:Npn \__debug_use_i_delimit_by_s_stop:nw #1 #2 \s__debug_stop {#1}
```

(End of definition for __debug_use_i_delimit_by_s_stop:nw.)

`\q__debug_recursion_tail` Internal quarks.

```
\q__debug_recursion_stop 41682 \quark_new:N \q__debug_recursion_tail
```

```
41683 \quark_new:N \q__debug_recursion_stop
```

(End of definition for \q__debug_recursion_tail and \q__debug_recursion_stop.)

`\__debug_if_recursion_tail_stop:N` Functions to query recursion quarks.

```
41684 \cs_new:Npn \__debug_use_none_delimit_by_q_recursion_stop:w
41685   #1 \q__debug_recursion_stop { }
41686 \__kernel_quark_new_test:N \__debug_if_recursion_tail_stop:N
```

(End of definition for __debug_if_recursion_tail_stop:N.)

`\debug_on:n`

`\debug_off:n`

`\__debug_all_on:`

`\__debug_all_off:`

```
41687 \cs_gset_protected:Npn \debug_on:n #1
41688   {
41689     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
41690     {
41691       \cs_if_exist_use:cF { __debug_ ##1 _on: }
41692       { \msg_error:nnn { debug } { debug } {##1} }
41693     }

```

```

41694     }
41695     \cs_gset_protected:Npn \debug_off:n #1
41696     {
41697         \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
41698         {
41699             \cs_if_exist_use:cF { __debug_ ##1 _off: }
41700             { \msg_error:nnn { debug } { debug } {##1} }
41701         }
41702     }
41703     \cs_new_protected:Npn \__debug_all_on:
41704     {
41705         \debug_on:n
41706         {
41707             check-assertions ,
41708             check-declarations ,
41709             check-expressions ,
41710             deprecation ,
41711             log-functions ,
41712         }
41713     }
41714     \cs_new_protected:Npn \__debug_all_off:
41715     {
41716         \debug_off:n
41717         {
41718             check-assertions ,
41719             check-declarations ,
41720             check-expressions ,
41721             deprecation ,
41722             log-functions ,
41723         }
41724     }

```

(End of definition for \debug_on:n and others. These functions are documented on page 31.)

\debug_suspend: Suspend and resume locally all debug-related errors and logging except deprecation errors.
\debug_resume: The \debug_suspend: and \debug_resume: pairs can be nested. We keep track of
 __debug_suspended:T nesting in a token list containing a number of periods. At first begin with the “non-
 \l__debug_suspended_tl suspended” version of __debug_suspended:T.

```

41725     \tl_new:N \l__debug_suspended_tl { }
41726     \cs_gset_protected:Npn \debug_suspend:
41727     {
41728         \tl_put_right:Nn \l__debug_suspended_tl { . }
41729         \cs_set_eq:NN \__debug_suspended:T \use:n
41730     }
41731     \cs_gset_protected:Npn \debug_resume:
41732     {
41733         \__kernel_tl_set:Nx \l__debug_suspended_tl
41734         { \tl_tail:N \l__debug_suspended_tl }
41735         \tl_if_empty:NT \l__debug_suspended_tl
41736         {
41737             \cs_set_eq:NN \__debug_suspended:T \use_none:n
41738         }
41739     }
41740     \cs_new_eq:NN \__debug_suspended:T \use_none:n

```

(End of definition for `\debug_suspend:` and others. These functions are documented on page 31.)

```

\__debug_check-assertions_on:
\__debug_check-assertions_off:
    \debug_assert:nN
    \debug_assert:nn
    \debug_assert:nNn
    \debug_assert:nnn
41741 \cs_new_protected:cpn { __debug_check-assertions_on: }
41742 {
41743     \cs_set:Npn \debug_assert:nN ##1##2
41744     {
41745         \__debug_suspended:T \use_none:nnnn
41746         \debug_assert:nNe
41747         { ##1 }
41748         ##2
41749         { Assertion~failed:~\tl_to_str:n { ##2 } }
41750     }
41751     \cs_set:Npn \debug_assert:nn ##1##2
41752     {
41753         \__debug_suspended:T \use_none:nnnn
41754         \debug_assert:nne
41755         { ##1 }
41756         { ##2 }
41757         { Assertion~failed:~\tl_to_str:n { ##2 } }
41758     }
41759     \cs_set:Npn \debug_assert:nNn ##1##2##3
41760     {
41761         \__debug_suspended:T \use_none:nnn
41762         \bool_if:NF ##2
41763         {
41764             \exp_args:Nne \__msg_expandable_error:nn
41765             { ##3 }
41766             { \msg_error_text:n { ##1 } }
41767         }
41768     }
41769     \cs_set:Npn \debug_assert:nnn ##1##2##3
41770     {
41771         \__debug_suspended:T \use_none:nnn
41772         \bool_if:NF { ##2 }
41773         {
41774             \exp_args:Nne \__msg_expandable_error:nn
41775             { ##3 }
41776             { \msg_error_text:n { ##1 } }
41777         }
41778     }
41779 }
41780 \cs_new_protected:cpn { __debug_check-assertions_off: }
41781 {
41782     \cs_set:Npn \debug_assert:nN ##1##2 { }
41783     \cs_set:Npn \debug_assert:nn ##1##2 { }
41784     \cs_set:Npn \debug_assert:nNn ##1##2##3 { }
41785     \cs_set:Npn \debug_assert:nnn ##1##2##3 { }
41786 }

```

(End of definition for `\__debug_check-assertions_on:` and others. These functions are documented on page 32.)

<pre> __debug_check-declarations_on: __debug_check-declarations_off: __kernel_chk_var_exist:N __kernel_chk_cs_exist:N __kernel_chk_cs_exist:c __kernel_chk_flag_exist:NN __kernel_chk_var_local:N __kernel_chk_var_global:N __kernel_chk_var_scope:NN </pre>	When debugging is enabled these two functions set up functions that test their argument (when <code>check-declarations</code> is active)

- `\__kernel_chk_var_exist:N` and `\__kernel_chk_cs_exist:N`, two functions that test that their argument is defined;
- `\__kernel_chk_var_scope:NN` that checks that its argument #2 has scope #1.
- `\__kernel_chk_var_local:N` and `\__kernel_chk_var_global:N` that perform both checks.

```

41787 \cs_new_protected:Npn \__kernel_chk_var_exist:N #1 { }
41788 \cs_new_protected:Npn \__kernel_chk_cs_exist:N #1 { }
41789 \cs_generate_variant:Nn \__kernel_chk_cs_exist:N { c }
41790 \cs_new:Npn \__kernel_chk_flag_exist:NN { }
41791 \cs_new_protected:Npn \__kernel_chk_var_local:N #1 { }
41792 \cs_new_protected:Npn \__kernel_chk_var_global:N #1 { }
41793 \cs_new_protected:Npn \__kernel_chk_var_scope:NN #1#2 { }
41794 \cs_new_protected:cpn { __debug_check-declarations_on: }
41795 {
41796   \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1
41797   {
41798     \__debug_suspended:T \use_none:nnn
41799     \cs_if_exist:NF ##1
41800     {
41801       \msg_error:nne { debug } { non-declared-variable }
41802       { \token_to_str:N ##1 }
41803     }
41804   }
41805   \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1
41806   {
41807     \__debug_suspended:T \use_none:nnn
41808     \cs_if_exist:NF ##1
41809     {
41810       \msg_error:nne { kernel } { command-not-defined }
41811       { \token_to_str:N ##1 }
41812     }
41813   }
41814   \cs_set:Npn \__kernel_chk_flag_exist:NN ##1##2
41815   {
41816     \__debug_suspended:T \use_iii:nnnn
41817     \flag_if_exist:NTF ##2
41818     { ##1 ##2 }
41819     {
41820       \msg_expandable_error:nnn { kernel } { bad-variable } {##2}
41821       ##1 \l_tmpa_flag
41822     }
41823   }
41824   \cs_set_protected:Npn \__kernel_chk_var_scope:NN
41825   {
41826     \__debug_suspended:T \use_none:nnn
41827     \__debug_chk_var_scope_aux:NN
41828   }
41829   \cs_set_protected:Npn \__kernel_chk_var_local:N ##1
41830   {
41831     \__debug_suspended:T \use_none:nnnnn
41832     \__kernel_chk_var_exist:N ##1
41833     \__debug_chk_var_scope_aux:NN 1 ##1

```

```

41834     }
41835     \cs_set_protected:Npn \__kernel_chk_var_global:N ##1
41836     {
41837         \__debug_suspended:T \use_none:nnnnn
41838         \__kernel_chk_var_exist:N ##1
41839         \__debug_chk_var_scope_aux:NN g ##1
41840     }
41841 }
41842 \cs_new_protected:cpn { __debug_check-declarations_off: }
41843 {
41844     \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1 { }
41845     \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1 { }
41846     \cs_set:Npn \__kernel_chk_flag_exist:NN { }
41847     \cs_set_protected:Npn \__kernel_chk_var_local:N ##1 { }
41848     \cs_set_protected:Npn \__kernel_chk_var_global:N ##1 { }
41849     \cs_set_protected:Npn \__kernel_chk_var_scope:NN ##1##2 { }
41850 }

```

(End of definition for `\__debug_check-declarations_on:` and others.)

`\__debug_chk_var_scope_aux:NN`
`\__debug_chk_var_scope_aux:Nn`
`\__debug_chk_var_scope_aux:NNn`

First check whether the name of the variable #2 starts with `<letter>_`. If it does then pass that letter, the `<scope>`, and the variable name to `\__debug_chk_var_scope_aux:NNn`. That function compares the two letters and triggers an error if they differ (the `\scan_stop:` case is not reachable here). If the second character was not `_` then pass the same data to the same auxiliary, except for its first argument which is now a control sequence. That control sequence is actually a token list (but to avoid triggering the checking code we manipulate it using `\cs_set_nopar:Npn`) containing a single letter `<scope>` according to what the first assignment to the given variable was.

```

41851 \cs_new_protected:Npn \__debug_chk_var_scope_aux:NN #1#2
41852 { \exp_args:NNf \__debug_chk_var_scope_aux:Nn #1 { \cs_to_str:N #2 } }
41853 \cs_new_protected:Npn \__debug_chk_var_scope_aux:Nn #1#2
41854 {
41855     \if:w _ \use_i:nn \__debug_use_i_delimit_by_s_stop:nw #2 ? ? \s__debug_stop
41856     \exp_after:wN \__debug_chk_var_scope_aux:NNn
41857     \__debug_use_i_delimit_by_s_stop:nw #2 ? \s__debug_stop
41858     #1 {#2}
41859     \else:
41860     \exp_args:Nc \__debug_chk_var_scope_aux:NNn
41861     { __debug_chk_/ #2 }
41862     #1 {#2}
41863     \fi:
41864 }
41865 \cs_new_protected:Npn \__debug_chk_var_scope_aux:NNn #1#2#3
41866 {
41867     \if:w #1 #2
41868     \else:
41869     \if:w #1 \scan_stop:
41870     \cs_gset_nopar:Npn #1 {#2}
41871     \else:
41872     \msg_error:nneee { debug } { local-global }
41873     {#1} {#2} { \iow_char:N \ \ #3 }
41874     \fi:
41875     \fi:
41876 }

```

```
41877 \use:c { __debug_check-declarations_off: }
```

(End of definition for `\__debug_chk_var_scope_aux:NN`, `\__debug_chk_var_scope_aux:Nn`, and `\__debug_chk_var_scope_aux:NNn`.)

`\__debug_log-functions_on:` These two functions (corresponding to the `expl3` option `log-functions`) control whether
`\__debug_log-functions_off:` `\__kernel_debug_log:e` writes to the log file or not. By default, logging is off.

```

41878 \cs_new_protected:cpn { __debug_log-functions_on: }
41879 {
41880   \cs_set_protected:Npn \__kernel_debug_log:e
41881     { \__debug_suspended:T \use_none:nn \iow_log:e }
41882 }
41883 \cs_new_protected:cpn { __debug_log-functions_off: }
41884 { \cs_set_protected:Npn \__kernel_debug_log:e { \use_none:n } }
41885 \cs_new_protected:Npn \__kernel_debug_log:e { \use_none:n }
```

(End of definition for `\__debug_log-functions_on:`, `\__debug_log-functions_off:`, and `\__kernel_debug_log:e`.)

`\__debug_check-expressions_on:` When debugging is enabled these two functions set `\__kernel_chk_expr:nNnN` to test or
`\__debug_check-expressions_off:` not whether the given expression is valid. The idea is to evaluate the expression within
`\__kernel_chk_expr:nNnN` a brace group (to catch trailing `\use_none:nn` or similar), then test that the result is
`\__debug_chk_expr_aux:nNnN` what we expect. This is done by turning it to an integer and hitting that with `\tex_`
`\__debug_chk_expr_aux:nNnN` `romannumeral:D` after replacing the first character by `-0`. If all goes well, that primitive
finds a non-positive integer and gives an empty output. If the original expression evaluation
stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation
(used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_`
`romannumeral:D` does not absorb `\tex_relax:D` the output will be nonempty. Note
that `#3` is empty except for mu expressions for which it is `\tex_mutoglue:D` to avoid
an “incompatible glue units” error. Note also that if we had omitted the first `\tex_`
`relax:D` then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer
expression.

```

41886 \cs_new_protected:cpn { __debug_check-expressions_on: }
41887 {
41888   \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2
41889     {
41890       \__debug_suspended:T { ##1 \use_none:nnnnnnn }
41891       \exp_after:wN \__debug_chk_expr_aux:nNnN
41892       \exp_after:wN { \tex_the:D ##2 ##1 \scan_stop: }
41893       ##2
41894     }
41895 }
41896 \cs_new_protected:cpn { __debug_check-expressions_off: }
41897 { \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2##3##4 {##1} }
41898 \cs_new:Npn \__kernel_chk_expr:nNnN #1#2#3#4 {#1}
41899 \cs_new:Npn \__debug_chk_expr_aux:nNnN #1#2#3#4
41900 {
41901   \tl_if_empty:oF
41902   {
41903     \tex_romannumeral:D - 0
41904     \exp_after:wN \use_none:n
41905     \int_value:w #3 #2 #1 \scan_stop:
41906   }
41907   {
```



```

41908         \msg_expandable_error:nnnn
41909         { debug } { expr } {#4} {#1}
41910     }
41911     #1
41912 }

```

(End of definition for `\__debug_check-expressions_on:` and others.)

`\__debug_deprecation_on:` Make deprecated commands throw errors if the user requests it. This relies on two token lists, filled up in `l3deprecation` by calls to `\__kernel_deprecation_code:nn`.

```

41913 \cs_new_protected:Npn \__debug_deprecation_on:
41914 { \g__debug_deprecation_on_tl }
41915 \cs_new_protected:Npn \__debug_deprecation_off:
41916 { \g__debug_deprecation_off_tl }

```

(End of definition for `\__debug_deprecation_on:` and `\__debug_deprecation_off:`.)

`\l__debug_tmp_tl` For patching.

```

\l__debug_tmpa_tl 41917 \tl_new:N \l__debug_tmp_tl
\l__debug_tmpb_tl 41918 \tl_new:N \l__debug_tmpa_tl
41919 \tl_new:N \l__debug_tmpb_tl

```

(End of definition for `\l__debug_tmp_tl`, `\l__debug_tmpa_tl`, and `\l__debug_tmpb_tl`.)

`\__debug_generate_parameter_list:NNN` Some functions don't take the arguments their signature indicates. For instance, `\clist_concat:NNN` doesn't take (directly) any argument, so patching it with something that uses `#1`, `#2`, or `#3` results in "Illegal parameter number in definition of `\clist_concat:NNN`".

`\__debug_arg_check_invalid:N` Instead of changing *the* definition of the macros, we'll create a copy of such macros, say, `\__debug_clist_concat:NNN` which will be defined as `<debug code with #1, #2 and #3>\clist_`. For that we need to identify the signature of every function and build the appropriate parameter list.

`\__debug_arg_terminate:w` `\__debug_generate_parameter_list:NNN` takes a function in `#1` and returns two parameter lists: `#2` contains the simple `#1#2#3` as would be used in the `<parameter text>` of the definition and `#3` contains the same parameters but with braces where necessary.

`\__debug_arg_if_braced:n` With the current implementation the resulting `#3` is, for example for `\some_function:NnNn`, `#1{#2}#3{#4}`. While this is correct, it might be unnecessary. Bracing everything will usually have the same outcome (unless the function was misused in the first place). What should be done?

```

41920 \cs_new_protected:Npn \__debug_generate_parameter_list:NNN #1#2#3
41921 {
41922     \__kernel_tl_set:Nx \l__debug_tmp_tl
41923     { \exp_last_unbraced:Nf \use_i:nnn \cs_split_function:N #1 }
41924     \__kernel_tl_set:Nx #2
41925     { \exp_args:NV \__debug_build_parm_text:n \l__debug_tmp_tl }
41926     \__kernel_tl_set:Nx #3
41927     { \exp_args:NV \__debug_build_arg_list:n \l__debug_tmp_tl }
41928 }
41929 \cs_new:Npn \__debug_build_parm_text:n #1
41930 {
41931     \__debug_arg_list_from_signature:nnN { 1 } \c_false_bool #1
41932     \q__debug_recursion_tail \q__debug_recursion_stop
41933 }

```

```

41934 \cs_new:Npn \__debug_build_arg_list:n #1
41935 {
41936   \__debug_arg_list_from_signature:nNN { 1 } \c_true_bool #1
41937   \q__debug_recursion_tail \q__debug_recursion_stop
41938 }
41939 \cs_new:Npn \__debug_arg_list_from_signature:nNN #1 #2 #3
41940 {
41941   \__debug_if_recursion_tail_stop:N #3
41942   \__debug_arg_check_invalid:N #3
41943   \bool_if:NT #2 { \__debug_arg_if_braced:NT #3 { \use_none:n } }
41944   \use:n { \c_hash_str \int_eval:n {#1} }
41945   \exp_args:Nf \__debug_arg_list_from_signature:nNN
41946     { \int_eval:n {#1+1} } #2
41947 }

```

Argument types w, p, T, and F shouldn't be included in the parameter lists, so we abort the loop if either is found.

```

41948 \cs_new:Npn \__debug_arg_check_invalid:N #1
41949 {
41950   \if:w w #1 \__debug_parm_terminate:w \else:
41951     \if:w p #1 \__debug_parm_terminate:w \else:
41952       \if:w T #1 \__debug_parm_terminate:w \else:
41953         \if:w F #1 \__debug_parm_terminate:w \else:
41954           \exp:w
41955         \fi:
41956       \fi:
41957     \fi:
41958   \fi:
41959   \exp_end:
41960 }
41961 \cs_new:Npn \__debug_parm_terminate:w
41962 { \exp_after:wN \__debug_use_none_delimit_by_q_recursion_stop:w \exp:w }
41963 \prg_new_conditional:Npnn \__debug_arg_if_braced:N #1 { T }
41964 { \exp_args:Nf \__debug_arg_if_braced:n { \__debug_get_base_form:N #1 } }
41965 \cs_new:Npn \__debug_arg_if_braced:n #1
41966 {
41967   \if:w n #1 \prg_return_true: \else:
41968     \if:w N #1 \prg_return_false: \else:
41969       \msg_expandable_error:nnn
41970         { debug } { bad-arg-type } {#1}
41971     \fi:
41972   \fi:
41973 }
41974 \msg_new:nnn { debug } { bad-arg-type }
41975 { Wrong~argument~type~#1. }

```

The macro below gets the base form of an argument type given a variant. It serves only to differentiate arguments which should be braced from ones which shouldn't. If all were to be braced this would be unnecessary. I moved the n and N variants to the beginning of the test as they are much more common here.

```

41976 \cs_new:Npn \__debug_get_base_form:N #1
41977 {
41978   \if:w n #1 \__debug_arg_return:N n \else:
41979     \if:w N #1 \__debug_arg_return:N N \else:
41980       \if:w c #1 \__debug_arg_return:N N \else:

```

```

41981         \if:w o #1 \__debug_arg_return:N n \else:
41982         \if:w V #1 \__debug_arg_return:N n \else:
41983         \if:w v #1 \__debug_arg_return:N n \else:
41984         \if:w f #1 \__debug_arg_return:N n \else:
41985         \if:w e #1 \__debug_arg_return:N n \else:
41986         \if:w x #1 \__debug_arg_return:N n \else:
41987         \__debug_arg_return:N \scan_stop:
41988         \fi:
41989         \fi:
41990         \fi:
41991         \fi:
41992         \fi:
41993         \fi:
41994         \fi:
41995         \fi:
41996         \fi:
41997         \exp_stop_f:
41998     }
41999 \cs_new:Npn \__debug_arg_return:N #1
42000 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w }

```

(End of definition for `\__debug_generate_parameter_list:NNN` and others.)

```

\__kernel_patch:nnn
\__kernel_patch_aux:nnn
\__debug_setup_debug_code:Nnn
\__debug_add_to_debug_code:Nnn
\__debug_insert_debug_code:Nnn
\__kernel_patch_weird:nnn
\__kernel_patch_weird_aux:nnn
\__debug_patch_weird:Nnn

```

Simple patching by adding material at the start and end of (a collection of) functions is straight-forward as we know the catcode set up. The approach is essentially that in `etoolbox`. Notice the need to worry about spaces: those are otherwise lost as normally in `expl3` code they would be `~`.

As discussed above, some functions don't take arguments, so we can't patch something that uses an argument in them. For these functions `\__kernel_patch:nnn` is used. It starts by creating a copy of the function (say, `\clist_concat:NNN`) with a `\__debug_` prefix in the name. This copy won't be changed. The code redefines the original function to take the exact same arguments as advertised in its signature (see `\__debug_generate_parameter_list:NNN` above). The redefined function also contains the debug code in the proper position. If a function with the same name and the `\__debug_` prefix was already defined, then the macro patches that definition by adding more debug code to it.

```

42001 \group_begin:
42002 \cs_set_protected:Npn \__kernel_patch:nnn
42003 {
42004     \group_begin:
42005     \char_set_catcode_other:N \#
42006     \__kernel_patch_aux:nnn
42007 }
42008 \cs_set_protected:Npn \__kernel_patch_aux:nnn #1#2#3
42009 {
42010     \char_set_catcode_parameter:N \#
42011     \char_set_catcode_space:N \ %
42012     \tex_endlinechar:D -1 \scan_stop:
42013     \tl_map_inline:nn {#3}
42014     {
42015         \cs_if_exist:cTF { \__debug_ \cs_to_str:N ##1 }
42016         { \__debug_add_to_debug_code:Nnn }
42017         { \__debug_setup_debug_code:Nnn }

```

```

42018         ##1 {#1} {#2}
42019     }
42020     \group_end:
42021 }
42022 \cs_set_protected:Npn \__debug_setup_debug_code:Nnn #1#2#3
42023 {
42024     \cs_gset_eq:cN { __debug_ \cs_to_str:N #1 } #1
42025     \__debug_generate_parameter_list:NNN #1 \l__debug_tmpa_tl \l__debug_tmpb_tl
42026     \exp_args:Ne \tex_scantokens:D
42027     {
42028         \tex_global:D \cs_prefix_spec:N #1
42029         \tex_def:D \exp_not:N #1
42030         \tl_use:N \l__debug_tmpa_tl
42031         {
42032             \tl_to_str:n {#2}
42033             \exp_not:c { __debug_ \cs_to_str:N #1 }
42034             \tl_use:N \l__debug_tmpb_tl
42035             \tl_to_str:n {#3}
42036         }
42037     }
42038 }
42039 \cs_set_protected:Npn \__debug_add_to_debug_code:Nnn #1#2#3
42040 {
42041     \use:e
42042     {
42043         \cs_set:Npn \exp_not:N \__debug_tmp:w
42044             ##1 \tl_to_str:n { macro: }
42045             ##2 \tl_to_str:n { -> }
42046             ##3 \c_backslash_str \tl_to_str:n { __debug_ }
42047                 \cs_to_str:N #1
42048             ##4 \s__debug_stop
42049         {
42050             \exp_not:N \exp_args:Ne \exp_not:N \tex_scantokens:D
42051             {
42052                 \tex_global:D ##1
42053                 \tex_def:D \exp_not:N #1 ##2
42054                 {
42055                     ##3 \tl_to_str:n {#2}
42056                     \c_backslash_str __debug_ \cs_to_str:N #1
42057                     ##4 \tl_to_str:n {#3}
42058                 }
42059             }
42060         }
42061     }
42062     \exp_after:wN \__debug_tmp:w \cs_meaning:N #1 \s__debug_stop
42063 }

```

Some functions, however, won't work with the signature reading setup above because their signature contains weird arguments. These functions need to be patched using `\__kernel_patch_weird:nnn`, which won't make a copy of the function, rather it will patch the debug code directly into it. This means that whatever argument the debug code uses must be actually used by the patched function.

```

42064 \cs_set_protected:Npn \__kernel_patch_weird:nnn
42065 {

```

```

42066     \group_begin:
42067     \char_set_catcode_other:N \#
42068     \__kernel_patch_weird_aux:nnn
42069   }
42070   \cs_set_protected:Npn \__kernel_patch_weird_aux:nnn #1#2#3
42071   {
42072     \char_set_catcode_parameter:N \#
42073     \char_set_catcode_space:N \ %
42074     \tex_endlinechar:D -1 \scan_stop:
42075     \tl_map_inline:nn {#3}
42076     { \__debug_patch_weird:Nnn ##1 {#1} {#2} }
42077   \group_end:
42078   }
42079   \cs_set_protected:Npn \__debug_patch_weird:Nnn #1#2#3
42080   {
42081     \use:e
42082     {
42083       \tex_endlinechar:D -1 \scan_stop:
42084       \exp_not:N \tex_scantokens:D
42085       {
42086         \tex_global:D \cs_prefix_spec:N #1
42087         \tex_def:D \exp_not:N #1
42088         \cs_parameter_spec:N #1
42089         {
42090           \tl_to_str:n {#2}
42091           \cs_replacement_spec:N #1
42092           \tl_to_str:n {#3}
42093         }
42094       }
42095     }
42096   }

```

(End of definition for __kernel_patch:nnn and others.)

Patching the second argument to ensure it exists. This happens before we alter #1 so the ordering is correct. For many variable types such as `int` a low-level error occurs when #2 is unknown, so adding a check is not needed.

```

42097   \__kernel_patch:nnn
42098   { \__kernel_chk_var_exist:N #2 }
42099   { }
42100   {
42101     \bool_set_eq:NN
42102     \bool_gset_eq:NN
42103     \clist_set_eq:NN
42104     \clist_gset_eq:NN
42105     \fp_set_eq:NN
42106     \fp_gset_eq:NN
42107     \prop_set_eq:NN
42108     \prop_gset_eq:NN
42109     \seq_set_eq:NN
42110     \seq_gset_eq:NN
42111     \str_set_eq:NN
42112     \str_gset_eq:NN
42113     \tl_set_eq:NN
42114     \tl_gset_eq:NN

```

```
42115 }
```

Patching both second and third arguments.

```
42116 \__kernel_patch:nnn
42117 {
42118     \__kernel_chk_var_exist:N #2
42119     \__kernel_chk_var_exist:N #3
42120 }
42121 { }
42122 {
42123     \clist_concat:NNN
42124     \clist_gconcat:NNN
42125     \prop_concat:NNN
42126     \prop_gconcat:NNN
42127     \seq_concat:NNN
42128     \seq_gconcat:NNN
42129     \str_concat:NNN
42130     \str_gconcat:NNN
42131     \tl_concat:NNN
42132     \tl_gconcat:NNN
42133 }
42134 \cs_gset_protected:Npn \__kernel_tl_set:Nx { \cs_set_nopar:Npe }
42135 \cs_gset_protected:Npn \__kernel_tl_gset:Nx { \cs_gset_nopar:Npe }
```

Patching where the first argument to a function needs scope-checking: either local or global (so two lists).

```
42136 \__kernel_patch:nnn
42137 { \__kernel_chk_var_local:N #1 }
42138 { }
42139 {
42140     \bool_set:Nn
42141     \bool_set_eq:NN
42142     \bool_set_true:N
42143     \bool_set_false:N
42144     \box_set_eq:NN
42145     \box_set_eq_drop:NN
42146     \box_set_to_last:N
42147     \clist_clear:N
42148     \clist_set_eq:NN
42149     \dim_zero:N
42150     \dim_set:Nn
42151     \dim_set_eq:NN
42152     \dim_add:Nn
42153     \dim_sub:Nn
42154     \fp_set_eq:NN
42155     \int_zero:N
42156     \int_set_eq:NN
42157     \int_add:Nn
42158     \int_sub:Nn
42159     \int_incr:N
42160     \int_decr:N
42161     \int_set:Nn
42162     \hbox_set:Nn
42163     \hbox_set_to_wd:Nnn
```

```

42164     \hbox_set:Nw
42165     \hbox_set_to_wd:Nnw
42166     \muskip_zero:N
42167     \muskip_set:Nn
42168     \muskip_add:Nn
42169     \muskip_sub:Nn
42170     \muskip_set_eq:NN
42171     \prop_clear:N
42172     \prop_concat:NNN
42173     \prop_pop:NnN
42174     \prop_pop:NnNT
42175     \prop_pop:NnNF
42176     \prop_pop:NnNTF
42177     \prop_put:Nnn
42178     \prop_put_if_not_in:Nnn
42179     \prop_put_from_keyval:Nn
42180     \prop_remove:Nn
42181     \prop_set_eq:NN
42182     \prop_set_from_keyval:Nn
42183     \seq_set_eq:NN
42184     \skip_zero:N
42185     \skip_set:Nn
42186     \skip_set_eq:NN
42187     \skip_add:Nn
42188     \skip_sub:Nn
42189     \str_clear:N
42190     \str_set_eq:NN
42191     \str_put_left:Nn
42192     \str_put_right:Nn
42193     \__kernel_tl_set:Nx
42194     \tl_clear:N
42195     \tl_set_eq:NN
42196     \tl_put_left:Nn
42197     \tl_put_left:Nv
42198     \tl_put_left:Nv
42199     \tl_put_left:Ne
42200     \tl_put_left:No
42201     \tl_put_right:Nn
42202     \tl_put_right:Nv
42203     \tl_put_right:Nv
42204     \tl_put_right:Ne
42205     \tl_put_right:No
42206     \tl_build_begin:N
42207     \tl_build_put_right:Nn
42208     \tl_build_put_left:Nn
42209     \vbox_set:Nn
42210     \vbox_set_top:Nn
42211     \vbox_set_to_ht:Nnn
42212     \vbox_set_top_to_ht:Nnn
42213     \vbox_set:Nw
42214     \vbox_set_to_ht:Nnw
42215     \vbox_set_split_to_ht:NNn
42216   }
42217   \__kernel_patch:nnn

```

```

42218 { \__kernel_chk_var_global:N #1 }
42219 { }
42220 {
42221   \bool_gset:Nn
42222   \bool_gset_eq:NN
42223   \bool_gset_true:N
42224   \bool_gset_false:N
42225   \box_gset_eq:NN
42226   \box_gset_eq_drop:NN
42227   \box_gset_to_last:N
42228   \cctab_gset:Nn
42229   \clist_gclear:N
42230   \clist_gset_eq:NN
42231   \dim_gset_eq:NN
42232   \dim_gzero:N
42233   \dim_gset:Nn
42234   \dim_gadd:Nn
42235   \dim_gsub:Nn
42236   \fp_gset_eq:NN
42237   \int_gzero:N
42238   \int_gset_eq:NN
42239   \int_gadd:Nn
42240   \int_gsub:Nn
42241   \int_gincr:N
42242   \int_gdecr:N
42243   \int_gset:Nn
42244   \hbox_gset:Nn
42245   \hbox_gset_to_wd:Nnn
42246   \hbox_gset:Nw
42247   \hbox_gset_to_wd:Nnw
42248   \muskip_gzero:N
42249   \muskip_gset:Nn
42250   \muskip_gadd:Nn
42251   \muskip_gsub:Nn
42252   \muskip_gset_eq:NN
42253   \prop_gclear:N
42254   \prop_gconcat:NNN
42255   \prop_gpop:NnN
42256   \prop_gpop:NnNT
42257   \prop_gpop:NnNF
42258   \prop_gpop:NnNTF
42259   \prop_gput:Nnn
42260   \prop_gput_if_not_in:Nnn
42261   \prop_gput_from_keyval:Nn
42262   \prop_gremove:Nn
42263   \prop_gset_eq:NN
42264   \prop_gset_from_keyval:Nn
42265   \seq_gset_eq:NN
42266   \skip_gzero:N
42267   \skip_gset:Nn
42268   \skip_gset_eq:NN
42269   \skip_gadd:Nn
42270   \skip_gsub:Nn
42271   \str_gclear:N

```



```

42272     \str_gset_eq:NN
42273     \str_gput_left:Nn
42274     \str_gput_right:Nn
42275     \__kernel_tl_gset:Nx
42276     \tl_gclear:N
42277     \tl_gset_eq:NN
42278     \tl_gput_left:Nn
42279     \tl_gput_left:Nv
42280     \tl_gput_left:Nv
42281     \tl_gput_left:Ne
42282     \tl_gput_left:No
42283     \tl_gput_right:Nn
42284     \tl_gput_right:Nv
42285     \tl_gput_right:Nv
42286     \tl_gput_right:Ne
42287     \tl_gput_right:No
42288     \tl_build_gbegin:N
42289     \tl_build_gput_right:Nn
42290     \tl_build_gput_left:Nn
42291     \vbox_gset:Nn
42292     \vbox_gset_top:Nn
42293     \vbox_gset_to_ht:Nnn
42294     \vbox_gset_top_to_ht:Nnn
42295     \vbox_gset:Nw
42296     \vbox_gset_to_ht:Nnw
42297     \vbox_gset_split_to_ht:Nnn
42298 }

```

Scoping for constants.

```

42299     \__kernel_patch:nnn
42300     { \__kernel_chk_var_scope:NN c #1 }
42301     { }
42302     {
42303         \bool_const:Nn
42304         \cctab_const:Nn
42305         \dim_const:Nn
42306         \int_const:Nn
42307         \intarray_const_from_clist:Nn
42308         \muskip_const:Nn
42309         \prop_const_from_keyval:Nn
42310         \prop_const_linked_from_keyval:Nn
42311         \skip_const:Nn
42312         \str_const:Nn
42313         \tl_const:Nn
42314     }

```

Flag functions.

```

42315     \__kernel_patch:nnn
42316     { \__kernel_chk_flag_exist:NN }
42317     { }
42318     {
42319         \flag_ensure_raised:N
42320         \flag_height:N
42321         \flag_if_raised:NT
42322         \flag_if_raised:NF

```

```

42323     \flag_if_raised:NTF
42324     \flag_if_raised_p:N
42325     \flag_raise:N
42326 }

```

Various one-offs.

```

42327 \__kernel_patch:nnn
42328 { \__kernel_chk_cs_exist:N #1 }
42329 { }
42330 { \cs_generate_variant:Nn }
42331 \__kernel_patch:nnn
42332 { \__kernel_chk_var_scope:NN g #1 }
42333 { }
42334 { \cctab_new:N }
42335 \__kernel_patch:nnn
42336 { \__kernel_chk_var_scope:NN l #1 }
42337 { }
42338 { \flag_new:N }
42339 \__kernel_patch:nnn
42340 {
42341     \__kernel_chk_var_scope:NN l #1
42342     \__kernel_chk_flag_exist:NN
42343 }
42344 { }
42345 { \flag_clear:N }
42346 \__kernel_patch:nnn
42347 { \__kernel_chk_var_scope:NN g #1 }
42348 { }
42349 { \intarray_new:Nn }
42350 \__kernel_patch:nnn
42351 { \__kernel_chk_var_scope:NN q #1 }
42352 { }
42353 { \quark_new:N }
42354 \__kernel_patch:nnn
42355 { \__kernel_chk_var_scope:NN s #1 }
42356 { }
42357 { \scan_new:N }

```

Patch various internal commands to log definitions of functions. First, a kernel internal. Then internals from the cs, keys and msg modules.

```

42358 \__kernel_patch:nnn
42359 { }
42360 {
42361     \__kernel_debug_log:e
42362     { Defining~\token_to_str:N #1~ \msg_line_context: }
42363 }
42364 { \__kernel_chk_if_free_cs:N }
42365 <@@=cs>
42366 \__kernel_patch_weird:nnn
42367 {
42368     \cs_if_free:NF #4
42369     {
42370         \__kernel_debug_log:e
42371         {
42372             Variant~\token_to_str:N #4~%

```

```

42373         already-defined;~ not~ changing~ it~ \msg_line_context:
42374     }
42375 }
42376 }
42377 { }
42378 { \__cs_generate_variant:wwNN }
42379 <@@=keys>
42380 \__kernel_patch:nnn
42381 {
42382     \cs_if_exist:cF { \c__keys_code_root_str #1 }
42383     { \__kernel_debug_log:e { Defining~key~#1~\msg_line_context: } }
42384 }
42385 { }
42386 { \__keys_cmd_set_direct:nn }
42387 <@@=msg>
42388 \__kernel_patch:nnn
42389 { }
42390 {
42391     \__kernel_debug_log:e
42392     { Defining~message~ #1 / #2 ~\msg_line_context: }
42393 }
42394 { \__msg_chk_free:nn }
42395 <@@=prg>

```

Internal functions from prg module.

```

42396 \__kernel_patch_weird:nnn
42397 { \__kernel_chk_cs_exist:c { #5 _p : #6 } }
42398 { }
42399 { \__prg_set_eq_conditional_p_form:wNnnnn }
42400 \__kernel_patch_weird:nnn
42401 { \__kernel_chk_cs_exist:c { #5 : #6 TF } }
42402 { }
42403 { \__prg_set_eq_conditional_TF_form:wNnnnn }
42404 \__kernel_patch_weird:nnn
42405 { \__kernel_chk_cs_exist:c { #5 : #6 T } }
42406 { }
42407 { \__prg_set_eq_conditional_T_form:wNnnnn }
42408 \__kernel_patch_weird:nnn
42409 { \__kernel_chk_cs_exist:c { #5 : #6 F } }
42410 { }
42411 { \__prg_set_eq_conditional_F_form:wNnnnn }
42412 <@@=regex>

```

Internal functions from regex module.

```

42413 \__kernel_patch:nnn
42414 {
42415     \__regex_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
42416     \group_begin:
42417         \__kernel_tl_set:Nx \l__regex_tmpa_tl
42418         { \__regex_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
42419         \use_none:nnn
42420     }
42421 { }
42422 { \__regex_escape_use:nnn }

```

```

42423 \__kernel_patch:nnn
42424 { \__regex_trace_push:nnN { regex } { 1 } \__regex_build:N }
42425 {
42426   \__regex_trace_states:n { 2 }
42427   \__regex_trace_pop:nnN { regex } { 1 } \__regex_build:N
42428 }
42429 { \__regex_build:N }
42430 \__kernel_patch:nnn
42431 { \__regex_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
42432 {
42433   \__regex_trace_states:n { 2 }
42434   \__regex_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
42435 }
42436 { \__regex_build_for_cs:n }
42437 \__kernel_patch:nnn
42438 {
42439   \__regex_trace:nne { regex } { 2 }
42440   {
42441     regex~new~state~
42442     L=\int_use:N \l__regex_left_state_int ~ -> ~
42443     R=\int_use:N \l__regex_right_state_int ~ -> ~
42444     M=\int_use:N \l__regex_max_state_int ~ -> ~
42445     \int_eval:n { \l__regex_max_state_int + 1 }
42446   }
42447 }
42448 { }
42449 { \__regex_build_new_state: }
42450 \__kernel_patch:nnn
42451 { \__regex_trace_push:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
42452 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
42453 { \__regex_group_aux:nnnnN }
42454 \__kernel_patch:nnn
42455 { \__regex_trace_push:nnN { regex } { 1 } \__regex_branch:n }
42456 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_branch:n }
42457 { \__regex_branch:n }
42458 \__kernel_patch:nnn
42459 {
42460   \__regex_trace_push:nnN { regex } { 1 } \__regex_match:n
42461   \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
42462 }
42463 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match:n }
42464 { \__regex_match:n }
42465 \__kernel_patch:nnn
42466 {
42467   \__regex_trace_push:nnN { regex } { 1 } \__regex_match_cs:n
42468   \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
42469 }
42470 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match_cs:n }
42471 { \__regex_match_cs:n }
42472 \__kernel_patch:nnn
42473 { \__regex_trace:nne { regex } { 1 } { initializing } }
42474 { }
42475 { \__regex_match_init: }
42476 \__kernel_patch:nnn

```

```

42477 {
42478   \__regex_trace:nne { regex } { 2 }
42479   { state~\int_use:N \l__regex_curr_state_int }
42480 }
42481 { }
42482 { \__regex_use_state: }
42483 \__kernel_patch:nnn
42484 { \__regex_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
42485 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
42486 { \__regex_replacement:n }
42487 \group_end:
42488 <@@=debug>

```

Patching arguments is a bit more involved: we do these one at a time. The basic idea is the same, using a # token that is a string.

```

42489 \group_begin:
42490 \cs_set_protected:Npn \__kernel_patch:Nn #1
42491 {
42492   \group_begin:
42493   \char_set_catcode_other:N \#
42494   \__kernel_patch_aux:Nn #1
42495 }
42496 \cs_set_protected:Npn \__kernel_patch_aux:Nn #1#2
42497 {
42498   \char_set_catcode_parameter:N \#
42499   \tex_endlinechar:D -1 \scan_stop:
42500   \exp_args:Ne \tex_scantokens:D
42501   {
42502     \tex_global:D \cs_prefix_spec:N #1 \tex_def:D \exp_not:N #1
42503     \cs_parameter_spec:N #1
42504     { \exp_args:No \tl_to_str:n { #1 #2 } }
42505   }
42506   \group_end:
42507 }

```

The functions here can get a bit repetitive, so we define a helper which can reuse the same patch code repeatedly. The main part of the patch is the same, so we just have to deal with the part which varies depending on the type of expression.

```

42508 \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
42509 {
42510   \tl_map_inline:nn {#1}
42511   {
42512     \exp_args:NNe \__kernel_patch:Nn ##1
42513     {
42514       { \c_hash_str 1 }
42515       {
42516         \exp_not:N \__kernel_chk_expr:nNn { \c_hash_str 2 }
42517         \exp_not:n {#2}
42518         \exp_not:N ##1
42519       }
42520     }
42521   }
42522 }
42523 <@@=dim>

```

```

42524 \__kernel_patch_eval:nn
42525 {
42526   \dim_set:Nn
42527   \dim_gset:Nn
42528   \dim_add:Nn
42529   \dim_gadd:Nn
42530   \dim_sub:Nn
42531   \dim_gsub:Nn
42532   \dim_const:Nn
42533 }
42534 { \__dim_eval:w { } }
42535 <@@=int>
42536 \__kernel_patch_eval:nn
42537 {
42538   \int_set:Nn
42539   \int_gset:Nn
42540   \int_add:Nn
42541   \int_gadd:Nn
42542   \int_sub:Nn
42543   \int_gsub:Nn
42544   \int_const:Nn
42545 }
42546 { \__int_eval:w { } }
42547 \__kernel_patch_eval:nn
42548 {
42549   \muskip_set:Nn
42550   \muskip_gset:Nn
42551   \muskip_add:Nn
42552   \muskip_gadd:Nn
42553   \muskip_sub:Nn
42554   \muskip_gsub:Nn
42555   \muskip_const:Nn
42556 }
42557 { \tex_muexpr:D { \tex_mutoglua:D } }
42558 \__kernel_patch_eval:nn
42559 {
42560   \skip_set:Nn
42561   \skip_gset:Nn
42562   \skip_add:Nn
42563   \skip_gadd:Nn
42564   \skip_sub:Nn
42565   \skip_gsub:Nn
42566   \skip_const:Nn
42567 }
42568 { \tex_glueexpr:D { } }

```

Patching expandable expressions, first the one-argument versions, then the two-argument ones.

```

42569 \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
42570 {
42571   \tl_map_inline:nn {#1}
42572   {
42573     \exp_args:NNe \__kernel_patch:Nn ##1
42574     {

```

```

42575         {
42576         \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
42577         \exp_not:n {#2}
42578         \exp_not:N ##1
42579       }
42580     }
42581   }
42582 }
42583 <@@=box>
42584   \__kernel_patch_eval:nn
42585   { \__box_dim_eval:n }
42586   { \__box_dim_eval:w { } }
42587 <@@=dim>
42588   \__kernel_patch_eval:nn
42589   {
42590     \dim_eval:n
42591     \dim_to_decimal:n
42592     \dim_to_decimal_in_sp:n
42593     \dim_abs:n
42594     \dim_sign:n
42595   }
42596   { \__dim_eval:w { } }
42597 <@@=int>
42598   \__kernel_patch_eval:nn
42599   {
42600     \int_eval:n
42601     \int_abs:n
42602     \int_sign:n
42603   }
42604   { \__int_eval:w { } }
42605   \__kernel_patch_eval:nn
42606   {
42607     \skip_eval:n
42608     \skip_horizontal:n
42609     \skip_vertical:n
42610   }
42611   { \tex_glueexpr:D { } }
42612   \__kernel_patch_eval:nn
42613   {
42614     \muskip_eval:n
42615   }
42616   { \tex_muexpr:D { \tex_mutogluue:D } }
42617   \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
42618   {
42619     \tl_map_inline:nn {#1}
42620     {
42621       \exp_args:NNe \__kernel_patch:Nn ##1
42622       {
42623         {
42624           \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
42625           \exp_not:n {#2}
42626           \exp_not:N ##1
42627         }
42628         {

```

```

42629         \exp_not:N \__kernel_chk_expr:nNn { \c_hash_str 2 }
42630         \exp_not:n {#2}
42631         \exp_not:N ##1
42632     }
42633 }
42634 }
42635 }
42636 <@@=dim>
42637 \__kernel_patch_eval:nn
42638 {
42639     \dim_max:nn
42640     \dim_min:nn
42641 }
42642 { \__dim_eval:w { } }
42643 <@@=int>
42644 \__kernel_patch_eval:nn
42645 {
42646     \int_max:nn
42647     \int_min:nn
42648     \int_div_truncate:nn
42649     \int_mod:nn
42650 }
42651 { \__int_eval:w { } }

Conditionals: three argument ones then one argument ones
42652 \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
42653 {
42654     \clist_map_inline:nn { :nNnT , :nNnF , :nNnTF , _p:nNn }
42655     {
42656         \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }
42657         {
42658             {
42659                 \exp_not:N \__kernel_chk_expr:nNn { \c_hash_str 1 }
42660                 \exp_not:n {#2}
42661                 \exp_not:c { #1 ##1 }
42662             }
42663             { \c_hash_str 2 }
42664             {
42665                 \exp_not:N \__kernel_chk_expr:nNn { \c_hash_str 3 }
42666                 \exp_not:n {#2}
42667                 \exp_not:c { #1 ##1 }
42668             }
42669         }
42670     }
42671 }
42672 <@@=dim>
42673 \__kernel_patch_cond:nn { dim_compare } { \__dim_eval:w { } }
42674 <@@=int>
42675 \__kernel_patch_cond:nn { int_compare } { \__int_eval:w { } }
42676 \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
42677 {
42678     \clist_map_inline:nn { :nT , :nF , :nTF , _p:n }
42679     {
42680         \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }
42681         {

```



```

42682         {
42683             \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
42684             \exp_not:n {#2}
42685             \exp_not:c { #1 ##1 }
42686         }
42687     }
42688 }
42689 }
42690 <@@=int>
42691 \__kernel_patch_cond:nn { int_if_even } { \__int_eval:w { } }
42692 \__kernel_patch_cond:nn { int_if_odd } { \__int_eval:w { } }

```

Step functions.

```

42693 <@@=dim>
42694 \__kernel_patch:Nn \dim_step_function:nnnN
42695 {
42696     {
42697         \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
42698         \dim_step_function:nnnN
42699     }
42700     {
42701         \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { }
42702         \dim_step_function:nnnN
42703     }
42704     {
42705         \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { }
42706         \dim_step_function:nnnN
42707     }
42708 }
42709 <@@=int>
42710 \__kernel_patch:Nn \int_step_function:nnnN
42711 {
42712     {
42713         \__kernel_chk_expr:nNnN {#1} \__int_eval:w { }
42714         \int_step_function:nnnN
42715     }
42716     {
42717         \__kernel_chk_expr:nNnN {#2} \__int_eval:w { }
42718         \int_step_function:nnnN
42719     }
42720     {
42721         \__kernel_chk_expr:nNnN {#3} \__int_eval:w { }
42722         \int_step_function:nnnN
42723     }
42724 }

```

Odds and ends

```

42725 \__kernel_patch:Nn \dim_to_fp:n { { (#1) } }
42726 \group_end:
42727 <@@=skip>

```

This one has catcode changes so must be done by hand.

```

42728 \cs_set_protected:Npn \__skip_tmp:w #1
42729 {

```

```

42730 \prg_set_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
42731 {
42732   \exp_after:wN \__skip_if_finite:wwNw
42733   \skip_use:N \tex_glueexpr:D
42734   \__kernel_chk_expr:nNnN
42735   {##1} \tex_glueexpr:D { } \skip_if_finite:n
42736   \__skip_sep: \prg_return_false:
42737   #1 \__skip_sep: \prg_return_true: \s__skip_stop
42738 }
42739 }
42740 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
42741 (@@=msg)

```

Messages.

```

42742 \msg_new:nnnn { debug } { debug }
42743 { The-debugging-option-~'##1'~does-not-exist~\msg_line_context:. }
42744 {
42745   The-functions-~'\iow_char:N\\debug_on:n'~and~
42746   '\iow_char:N\\debug_off:n'~only-accept-the-arguments~
42747   'all',~'check-declarations',~'check-expressions',~
42748   'deprecation',~'log-functions',~not~'##1'.
42749 }
42750 \msg_new:nnn { debug } { expr } { '##2'~in~##1 }
42751 \msg_new:nnnn { debug } { local-global }
42752 { Inconsistent-local/global~assignment }
42753 {
42754   \c__msg_coding_error_text_tl
42755   \if:w l #2 Local
42756   \else:
42757     \if:w g #2 Global \else: Constant \fi:
42758   \fi:
42759   \ %
42760   assignment~to~a~
42761   \if:w l #1 local
42762   \else:
42763     \if:w g #1 global \else: constant \fi:
42764   \fi:
42765   \ %
42766   variable-~'##3'.
42767 }
42768 \msg_new:nnnn { debug } { non-declared-variable }
42769 { The-variable-##1~has-not-been-declared~\msg_line_context:. }
42770 {
42771   \c__msg_coding_error_text_tl
42772   Checking-is~active,~and-you-have-tried-do-so~something~like: \\
42773   \\ \tl_set:Nn ~ ##1 ~ \{ ~ ... ~ \} \\
42774   without-first~having: \\
42775   \\ \tl_new:N ~ ##1 \\
42776   \\
42777   LaTeX~will~continue,~creating~the~variable~where~it~is~the~one~being~set.
42778 }

```

__kernel_if_debug:TF Flip the switch for deprecated code.

```

42779 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#1}

```

(End of definition for _kernel_if_debug:TF.)

42780 **</def>**

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
$\backslash!$	20776, 20792
$!$	281
$\backslash"$	19964, 19967, 33333, 35799, 35817, 35841, 35847, 35851, 35857, 35861, 35867, 35873, 35880, 35881, 35887, 35891, 35893, 36002
$\backslash\#$..	10749, 14425, 19964, 35736, 42005, 42010, 42067, 42072, 42493, 42498
$\backslash\$$	5347, 14424, 19964, 19967, 35736
$\backslash\%$	10751, 14426, 19964, 35736
$\backslash\&$	9448, 14417, 19964, 19967
$\&\&$	280
$\backslash'$	33333, 35799, 35811, 35838, 35845, 35849, 35854, 35859, 35862, 35864, 35871, 35876, 35877, 35884, 35889, 35892, 35900, 35901, 35948, 35949, 35956, 35957, 35968, 35969, 35974, 35975, 36003, 36004, 36026, 36027, 36030, 36031, 36032, 36033
$\backslash($	32943
$\backslash)$	32943
$\backslash*$	9210, 9222, 14013, 14036, 20146, 20148, 20152, 20160
$*$	281
$**$	281
$+$	281
$\backslash,$	22348, 35748
$\backslash-$	149
$-$	281
$\backslash.$	33333, 35799, 35816, 35904, 35905, 35914, 35915, 35924, 35925, 35942, 35954, 35955, 36005, 36006, 36036, 36037, 36040, 36041
$\backslash/$	148, 4229
$/$	281
$\backslash:$	14423
$\backslash::$	45, 424, 443, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2455, 2457, 2458, 2459, 2466, 2469, 2472, 2478, 2634, 2636, 2641, 2646, 2648, 2653, 2705, 2706, 2707, 2708, 2709, 2720
$\backslash::N$	45, 2451, 2709
$\backslash::V$	45, 2472
$\backslash::V_{\text{unbraced}}$	45, 2633
$\backslash::c$	45, 2453
$\backslash::e$	45, 2457
$\backslash::e_{\text{unbraced}}$	45, 2633
$\backslash::f$	45, 2459, 2708
$\backslash::f_{\text{unbraced}}$	45, 2633, 2706
$\backslash::n$	45, 853, 2450, 2705, 2706, 2709
$\backslash::o$	45, 2455, 2707
$\backslash::o_{\text{unbraced}}$	45, 2633, 2705, 2707, 2708, 2709
$\backslash::p$	45, 424, 2452
$\backslash::v$	45, 2472
$\backslash::v_{\text{unbraced}}$	45, 2633
$\backslash::x$	45, 2466
$\backslash::x_{\text{unbraced}}$	45, 2633, 2720
$<$	281
$\backslash=$	22349, 33333, 35799, 35814, 35894, 35895, 35910, 35911, 35933, 35934, 35935, 35962, 35963, 35988, 35989, 36042, 36043
$=$	281
$>$	281
$?$	281
$?:$	280
$\backslash???$	91, 654
$\backslash\backslash$	2304, 2409, 3559, 3562, 3563, 3587, 3588, 3595, 3596, 4130, 4360, 4684, 4685, 6081, 6088, 6089, 6090, 6214, 8040, 8044, 8049, 8083, 8092, 8096, 8101, 8121, 8123, 8124, 8126, 8129, 8131, 8136, 8138, 8140, 8145, 8149, 8152, 8156, 8158, 8162, 8164, 8170, 8172, 8176, 8178, 8182, 8187, 8189, 8231, 8233, 8238, 8240, 8246, 8251, 8252, 8256, 8260, 8270, 8273, 8277, 8278, 8282, 8290, 8316, 8361, 9351, 9369, 9371, 9376, 9377, 9401, 9411, 9418, 9433, 9854, 9862, 9869, 9881, 9882, 9897, 9898, 9905, 9919, 9922, 9923, 9955, 9990, 10023, 10024, 10037, 10094, 10095, 10103, 10118, 10119, 10148, 10159, 10163, 10169, 10176, 10199, 10350, 10754, 11738, 11742, 11743, 11745, 11751, 11753, 11758, 11759, 11761, 11762, 11764, 11766, 11778, 11788, 11790, 11791, 11792, 11888, 11889, 14419, 14976, 14977, 14980, 15303,

- 15306, 15307, 15308, 15309, 15314,
15320, 15325, 15332, 15491, 15494,
15495, 15496, 15498, 15504, 15509,
15514, 15665, 15672, 19964, 23732,
23750, 23756, 24766, 24769, 24770,
24771, 24778, 24781, 24782, 31492,
31493, 31500, 31876, 31878, 31879,
31882, 31884, 31885, 31888, 31890,
31891, 31892, 31896, 31903, 35734,
38621, 40275, 40277, 40304, 40306,
40326, 40328, 40349, 40351, 40358,
40360, 40367, 40369, 40375, 40388,
40390, 40804, 41873, 42745, 42746,
42772, 42773, 42774, 42775, 42776
- $\backslash\{$ 4633, 8044, 8049,
8096, 8138, 8140, 8152, 8189, 8278,
8282, 10748, 14420, 14981, 19964,
32983, 32984, 32985, 35736, 42773
- $\backslash}$ 64, 8043, 8049, 8153, 8189, 8278, 8282,
10750, 14421, 14981, 19964, 32983,
32984, 32985, 32986, 35736, 42773
- $\backslashsqcup$ 64, 66, 69, 71, 147, 1802, 3609, 3810,
4190, 4578, 4583, 4627, 4637, 4822,
7300, 9197, 9899, 10080, 10755,
11762, 14013, 14036, 14981, 15306,
15307, 15308, 19964, 20086, 20089,
31919, 31925, 31936, 31962, 32452,
32460, 32981, 32982, 35747, 42011,
42073, 42759, 42765, 42773, 42775
- $\backslash$... 59, 1956, 2743, 3669, 3689, 3766,
3769, 4579, 4584, 4585, 4586, 4587,
4590, 4601, 4638, 4692, 4694, 4696,
4698, 4700, 4702, 5346, 7251, 7254,
7268, 7271, 7280, 7283, 7286, 7289,
7303, 7306, 8706, 8709, 9455, 10659,
10702, 12639, 14422, 15095, 15096,
15432, 15433, 15614, 15615, 15616,
19964, 19967, 19969, 19975, 20023,
28770, 33333, 35799, 35812, 35839,
35846, 35850, 35855, 35860, 35865,
35872, 35878, 35879, 35885, 35890,
35902, 35903, 35920, 35921, 35928,
35929, 35943, 35944, 35945, 35976,
35977, 35998, 35999, 36000, 36001
- $\wedge$ 281
- $\backslash_$ 14428, 19964, 19967, 35736
- $\backslash^$ 33333,
35799, 35810, 35837, 35844, 35848,
35853, 35858, 35863, 35870, 35874,
35875, 35883, 35888, 36028, 36029
- $\parallel$ 280
- $\backslash\sim$ 64, 4623, 4627, 4633, 10752,
12538, 12583, 14427, 19964, 19967,
33333, 35740, 35799, 35813, 35840,
35852, 35856, 35866, 35882, 35886,
35930, 35931, 35932, 35986, 35987
- ### A
- $\backslash A$ 14014, 14037
- $\backslash AA$ 33337, 34878, 35760
- $\backslash aa$ 33337, 34878, 35770
- $\backslash above$ 150
- $\backslash abovedisplaysshortskip$ 151
- $\backslash abovedisplayskip$ 152
- $\backslash abovewithdelims$ 153
- abs 281
- $\backslash accent$ 154
- $acos$ 284
- $acosc$ 284
- $acot$ 285
- $acotd$ 285
- $acsc$ 284
- $acscd$ 284
- $\backslash adjdemerits$ 155
- $\backslash adjustspacing$ 932
- $\backslash advance$ 156
- $\backslash AE$ 33338, 34879, 35761, 36030
- $\backslash ae$ 33338, 34879, 35771, 36031
- $\backslash afterassignment$ 157
- $\backslash aftergroup$ 158
- $\backslash alignmark$ 779
- $\backslash aligntab$ 780
- $asec$ 284
- $asecd$ 284
- $asin$ 284
- $asind$ 284
- $atan$ 285
- $atand$ 285
- $\backslash AtBeginDocument$ 696, 11650
- $\backslash atop$ 159
- $\backslash atopwithdelims$ 160
- $\backslash attribute$ 781
- $\backslash attributedef$ 782
- $\backslash automaticdiscretionary$ 783
- $\backslash automatichyphenmode$ 785
- $\backslash automatichyphenpenalty$ 786
- $\backslash autospacing$ 1134
- $\backslash autoxspacing$ 1135
- ### B
- $\backslash b$ 33333, 35799, 35824
- $\backslash babelshorthand$ 32938
- $\backslash badness$ 161
- $\backslash baselineskip$ 162
- $\backslash batchmode$ 163
- $\backslash begin$.. 10200, 32936, 32946, 35731, 40532
- $\backslash beginscname$ 788
- $\backslash begingroup$ 3, 7, 12, 16, 35, 63, 68, 142, 164

- \beginL 472
- \beginR 473
- \belowdisplaysshortskip 165
- \belowdisplayskip 166
- benchmark commands:
 - \benchmark:n
 - 348, 349, 1546, 41247, 41247
 - \g_benchmark_duration_target_fp .
 - 348, 1545, 1547, 41173, 41306, 41312
 - \benchmark_once:n .. 348, 41235, 41235
 - \benchmark_once_silent:n
 - 348, 41235, 41237, 41240
 - \g_benchmark_ops_fp
 - 348, 1546, 41225, 41245, 41257, 41330
 - \benchmark_silent:n 348, 41249, 41252
 - \benchmark_tic:
 - 348, 349, 41341, 41341, 41368
 - \g_benchmark_time_fp 348,
 - 1546, 41225, 41244, 41245, 41257,
 - 41297, 41310, 41329, 41356, 41362
 - \benchmark_toc:
 - 348, 349, 41347, 41347, 41367
- benchmark internal commands:
 - __benchmark_aux:
 - 1547, 1549, 41256, 41259, 41259, 41309
 - \g__benchmark_code_tl 1549,
 - 41234, 41255, 41262, 41301, 41307
 - __benchmark_display:
 - 41238, 41250, 41325, 41325
 - \g__benchmark_duration_int
 - 1548, 41227, 41263,
 - 41267, 41279, 41305, 41308, 41311
 - __benchmark_fp_to_tl:N
 - .. 41314, 41314, 41329, 41330, 41362
 - __benchmark_fp_to_tl_aux:nN ...
 - 41314, 41319, 41323
 - __benchmark_measure_op:
 - 41242, 41254, 41303, 41303
 - \g__benchmark_nesting_int 41175
 - \g__benchmark_one_op_fp
 - 41245, 41257, 41302, 41310
 - __benchmark_raw:nN . 1545, 41175,
 - 41176, 41215, 41221, 41243, 41262
 - __benchmark_raw_aux: . 41181, 41188
 - __benchmark_raw_aux:N
 - 41175, 41179, 41185
 - __benchmark_raw_end:N
 - 41175, 41183, 41190
 - __benchmark_raw_replicate:nnN ..
 - 41199, 41200, 41209, 41301
 - __benchmark_raw_replicate_-
 - aux:nnN 41216, 41219
 - __benchmark_raw_replicate_-
 - large:nnN 41203, 41206
 - __benchmark_raw_replicate_-
 - small:nnN 41204, 41212
 - \g__benchmark_repeat_int
 - 1548, 41233, 41261, 41268, 41271,
 - 41274, 41279, 41283, 41298, 41301
 - __benchmark_run:N 41272,
 - 41285, 41286, 41287, 41288, 41300
 - \g__benchmark_tictoc_int
 - 41333, 41339, 41345, 41355
 - \l__benchmark_tictoc_pop_tl
 - 41333, 41349, 41357
 - __benchmark_tictoc_prefix:
 - 41336, 41336, 41343, 41360
 - \g__benchmark_tictoc_seq
 - 41333, 41344, 41349
 - \g__benchmark_time_a_int
 - 41228, 41284, 41285, 41292
 - \g__benchmark_time_b_int
 - 41228, 41286, 41292
 - \g__benchmark_time_c_int
 - 41228, 41287, 41293
 - \g__benchmark_time_d_int
 - 41228, 41288, 41293
 - \g__benchmark_time_int ... 41215,
 - 41217, 41221, 41222, 41228, 41243,
 - 41244, 41262, 41263, 41267, 41272,
 - 41280, 41284, 41289, 41296, 41298
 - __benchmark_tmp:w
 - 1546, 41199, 41199, 41208,
 - 41210, 41214, 41215, 41221, 41223
 - __benchmark_toc: 41347, 41350, 41353
- \binoppenalty 167
- bitset commands:
 - \bitset_addto_named_index:Nn ...
 - 293, 31321, 31321
 - \bitset_clear:N
 - 294, 31411, 31411, 31419
 - \bitset_gclear:N
 - 294, 31411, 31415, 31420
 - \bitset_gset_false:Nn
 - 294, 31377, 31383, 31410
 - \bitset_gset_true:Nn
 - 294, 31377, 31379, 31408
 - \bitset_if_exist:N 31327, 31329
 - \bitset_if_exist:NTF 294, 31326
 - \bitset_if_exist_p:N 294, 31326
 - \bitset_item:Nn
 - 294, 31443, 31443, 31458
 - \bitset_log:N 295, 31459, 31461, 31462
 - \bitset_log_named_index:N
 - 295, 31474, 31477, 31479
 - \bitset_new:N 293, 31304, 31304, 31319
 - \bitset_new:Nn 293, 31304, 31310, 31320

- \bitset_set_false:Nn
..... [294](#), [31377](#), 31381, 31409
- \bitset_set_true:Nn
..... [294](#), [31377](#), 31377, 31407
- \bitset_show:N
..... [294](#), [295](#), [31459](#), 31459, 31460
- \bitset_show_named_index:N
..... [295](#), [31474](#), 31474, 31476
- \bitset_to_arabic:N [292](#),
[295](#), [1303](#), [31421](#), 31421, 31439, 31470
- \bitset_to_bin:N
[293](#), [295](#), [31421](#), 31435, 31440, 31469
- \bitset_use:N [295](#), [31441](#), 31441, 31442
- bitset internal commands:
- _bitset_gset_false:Nn
..... [31330](#), 31336, 31384
- _bitset_gset_true:Nn
..... [31330](#), 31332, 31380
- _bitset_set:NNn
 .. [31378](#), 31380, 31382, 31384, 31385
- _bitset_set:NNnN [31330](#),
 [31331](#), [31333](#), [31335](#), [31337](#), 31338
- _bitset_set_aux:NNn [31377](#)
- _bitset_set_false:Nn
..... [31330](#), 31334, 31382
- _bitset_set_true:Nn
..... [31330](#), 31330, 31378
- _bitset_show:NN [31459](#), 31461, 31463
- _bitset_show_named_index:NN ...
..... [31475](#), 31478, 31480
- _bitset_test_digits:n [31362](#)
- _bitset_test_digits:nTF
..... [31362](#), 31395
- _bitset_test_digits:w
..... [31362](#), 31364, 31376
- _bitset_test_digits_end:
..... [31366](#), 31368, 31375, 31376
- _bitset_test_digits_end:n .. [31362](#)
- _l_bitset_tmp_int [31361](#), 31365, 31369
- _bitset_to_int:nN
..... [31421](#), 31426, 31430, 31433
- \bodydir [789](#)
- \bodydirection [790](#)
- bool commands:
- \bool_case:n
..... [73](#), [8627](#), 8633, 41486, 41487
- \bool_case:nTF
 [73](#), [8627](#), 8627, 8629, 8631, 41488,
 41489, 41490, 41491, 41492, 41493
- \bool_case_true:n [41486](#), 41487
- \bool_case_true:nTF
..... [41486](#), 41489, 41491, 41493
- \bool_const:Nn
..... [68](#), [8370](#), 8370, 8375, 42303
- \bool_do_until:Nn
..... [72](#), [8593](#), 8595, 8596, 8598
- \bool_do_until:nn [72](#), [8599](#), 8620, 8623
- \bool_do_while:Nn
..... [72](#), [8593](#), 8593, 8594, 8597
- \bool_do_while:nn [72](#), [8599](#), 8607, 8610
- .bool_gset:N [247](#), [22942](#)
- \bool_gset:Nn
..... [68](#), [8392](#), 8397, 8403, 42221
- \bool_gset_eq:NN [68](#), 4569,
 [6735](#), [8388](#), 8389, 8391, 42102, 42222
- \bool_gset_false:N [68](#), 6683, [8376](#),
 8382, 8387, 8408, 14633, 14642, 42224
- .bool_gset_inverse:N [247](#), [22950](#)
- \bool_gset_inverse:N
..... [69](#), [8404](#), 8407, 8409
- \bool_gset_true:N
..... [68](#), 6748, [8376](#), 8380,
 8386, 8408, 8943, 14623, 40881,
 40902, 40982, 41065, 41482, 42223
- \bool_if:N [8415](#), 8423
- \bool_if:n [8466](#)
- \bool_if:NTF [69](#), 108,
 2148, 5462, 5471, 5912, 6085, 6171,
 6189, 6207, 6358, 6577, 6585, 6817,
 7427, 7450, 7523, 7750, 7917, 7923,
 7964, 8334, 8339, 8405, 8408, [8415](#),
 8426, 8485, 8588, 8590, 8594, 8596,
 8941, 10970, 10977, 14637, 14646,
 20999, 22385, 22495, 22596, 22853,
 22862, 22912, 23165, 23288, 23339,
 23355, 23357, 23362, 23369, 23432,
 23452, 23459, 23465, 23500, 23510,
 23538, 33785, 37790, 38492, 38833,
 40645, 40886, 41114, 41762, 41943
- \bool_if:nTF
..... [68](#), [71](#)–[73](#), [954](#), 6174, 8432,
 [8466](#), 8538, 8545, 8564, 8571, 8580,
 8601, 8610, 8614, 8623, 8642, 8720,
 11847, 12233, 12238, 17492, 41772
- \bool_if_exist:N [8462](#), 8464
- \bool_if_exist:NTF .. [69](#), [8462](#), 22637
- \bool_if_exist_p:N [69](#), [8462](#)
- \bool_if_p:N [69](#), [8415](#)
- \bool_if_p:n
 . [71](#), [611](#), 8373, 8395, 8400, [8466](#),
 [8474](#), 8474, 8545, 8571, 8577, 8581
- \bool_lazy_all:n [8527](#)
- \bool_lazy_all:nTF
..... [70](#), [71](#), 5745, [8525](#), 41137
- \bool_lazy_all_p:n [71](#), [8525](#)
- \bool_lazy_and:nn [8542](#)
- \bool_lazy_and:nnTF
..... [70](#), [71](#), 3633, [8542](#),

- 8795, 9081, 10512, 11723, 31697,
- 32678, 32947, 33101, 33210, 33272,
- 33816, 33962, 34696, 34753, 34790,
- 35046, 35086, 35646, 37501, 38404,
- 38815, 40874, 41075, 41146, 41158
- \bool_lazy_and_p:nn
 . 71, 8542, 33775, 34837, 41086, 41098
- \bool_lazy_any:n 8553
- \bool_lazy_any:nTF 70, 71,
 8551, 11331, 14475, 14749, 14773,
 14795, 14965, 32812, 32961, 34485
- \bool_lazy_any_p:n
 71, 8551, 33104, 34703
- \bool_lazy_or:nn 8568
- \bool_lazy_or:nnTF 70, 71,
 3675, 3697, 8568, 8778, 8906, 11274,
 32714, 33263, 33410, 33772, 34059,
 34114, 34235, 34551, 34772, 34835,
 35172, 35211, 35658, 35697, 36136,
 36228, 36265, 36291, 36295, 38835,
 39220, 40852, 41083, 41095, 41154
- \bool_lazy_or_p:nn 71, 8568, 33275,
 33965, 34698, 34755, 34793, 35649
- \bool_log:N 69, 8439, 8441, 8442
- \bool_log:n 69, 8435, 8437
- \bool_new:N .. 68, 4428, 4869, 6652,
 6653, 6655, 6656, 6657, 8368, 8368,
 8369, 8458, 8459, 8460, 8461, 8938,
 10697, 14486, 22224, 22462, 22467,
 22468, 22475, 22476, 22481, 22484,
 22637, 33353, 37376, 38646, 40873
- \bool_not_p:n 71, 8577, 8577
- .bool_set:N 247, 22942
- \bool_set:Nn
 68, 603, 607, 8392, 8392, 8402, 42140
- \bool_set_eq:NN 68, 4563, 6896, 8388,
 8388, 8390, 21005, 21007, 42101, 42141
- \bool_set_false:N
 68, 122, 5436, 5641,
 6627, 6698, 6712, 6774, 6816, 8376,
 8378, 8385, 8405, 10803, 10946,
 10954, 10962, 10972, 10979, 22537,
 23187, 23188, 23189, 23199, 23200,
 23215, 23235, 23236, 23255, 23265,
 23346, 23428, 37783, 38490, 42143
- .bool_set_inverse:N 247, 22950
- \bool_set_inverse:N
 69, 8404, 8404, 8406
- \bool_set_true:N
 .. 68, 135, 5441, 5645, 6621, 6814,
 6895, 8376, 8376, 8384, 8405, 10932,
 22532, 23198, 23216, 23217, 23237,
 23252, 23260, 23351, 23444, 33354,
 37804, 37826, 37857, 39408, 42142
- \bool_show:N 69, 8439, 8439, 8440
- \bool_show:n 69, 8435, 8435
- \bool_to_str:N .. 69, 8424, 8424, 8429
- \bool_to_str:n
 69, 8424, 8430, 8436, 8438
- \bool_until_do:Nn
 72, 8587, 8589, 8590, 8592
- \bool_until_do:nn
 72, 8599, 8612, 8617, 41265
- \bool_while_do:Nn
 72, 8587, 8587, 8588, 8591
- \bool_while_do:nn 73, 8599, 8599, 8604
- \bool_xor:nn 8578
- \bool_xor:nnTF 72, 8578
- \bool_xor_p:nn 72, 8578
- \c_false_bool 68, 69,
 406, 436, 590, 604, 607–609, 1651,
 1703, 1704, 1735, 1759, 1764, 1796,
 1815, 2085, 2092, 2800, 3060, 5124,
 5142, 5334, 5381, 5680, 5882, 5899,
 5912, 6108, 6244, 6745, 7852, 7861,
 7870, 7880, 7942, 7950, 8368, 8379,
 8383, 8450, 8485, 8516, 8539, 8545,
 8563, 8731, 21001, 22876, 22878,
 22885, 22890, 40889, 41142, 41931
- \g_tmpa_bool 70, 8458
- \l_tmpa_bool 69, 8458
- \g_tmpb_bool 70, 8458
- \l_tmpb_bool 69, 8458
- \c_true_bool
 .. 68, 69, 406, 604, 607–609, 731,
 1703, 1735, 1796, 1814, 2106, 4435,
 4566, 5007, 5081, 5138, 5324, 5326,
 5328, 5330, 5332, 5342, 5380, 5387,
 5880, 5890, 5912, 5913, 6106, 6227,
 6229, 6252, 6344, 6515, 6526, 6541,
 6702, 7474, 7591, 7704, 8377, 8381,
 8449, 8485, 8517, 8518, 8537, 8565,
 8571, 8638, 8725, 21000, 21005,
 22883, 22892, 41142, 41554, 41936
- bool internal commands:
- __bool_!:Nw 8496
- __bool_&_0: 8508
- __bool_&_1: 8508
- __bool_&_2: 8508
- __bool_(:Nw 8501
- __bool_)_0: 8508
- __bool_)_1: 8508
- __bool_)_2: 8508
- __bool_case:NnTF 8627
- __bool_case:nTF
 8628, 8630, 8632, 8634, 8635
- __bool_case:w 8627, 8637, 8640, 8644
- __bool_case_end:nw 8643, 8646

- _bool_choose:NNN 8503, 8507, 8508, 8508
- _bool_get_next:NN 608, 8482, 8486, 8486, 8498, 8504, 8519, 8520, 8521, 8522, 8523, 8524
- _bool_if_p:n 8474, 8474, 8475
- _bool_if_p_aux:w 607, 8474, 8477, 8484
- _bool_if_recursion_tail_stop-do:nn 8414, 8414, 8537, 8563
- _bool_lazy_all:n 8525, 8526, 8535, 8540
- _bool_lazy_any:n 8551, 8552, 8561, 8566
- _bool_p:Nw 8506
- _bool_show:NN 8439, 8439, 8441, 8443
- _bool_use_i_delimit_by_q-recursion_stop:nw 8412, 8412, 8539, 8565
- _bool_|_0: 8508
- _bool_|_1: 8508
- _bool_|_2: 8508
- \botmark 168
- \botmarks 474
- \boundary 791
- \box 169
- box commands:
 - \box_autosize_to_wd_and_ht:Nnn .. 320, 37091, 37091, 37093
 - \box_autosize_to_wd_and_ht_plus_dp:Nnn ... 320, 37091, 37097, 37102
 - \box_clear:N 311, 312, 36398, 36398, 36402, 36405, 37409, 37496, 37573
 - \box_clear_new:N 312, 36404, 36404, 36408
 - \box_dp:N 312, 1422, 25256, 36426, 36427, 36430, 36433, 36438, 36442, 36662, 36723, 36836, 36965, 37080, 37099, 37105, 37187, 37199, 37229, 37260, 37267, 37272, 37610, 37611, 37729, 37734, 37762, 37776, 37948, 38226, 38247, 38557
 - \box_frame:Nnn 322, 37168, 37168, 37173
 - \box_gautosize_to_wd_and_ht:Nnn . 320, 37091, 37094, 37096
 - \box_gautosize_to_wd_and_ht-plus_dp:Nnn 320, 37091, 37103, 37108
 - \box_gclear:N 311, 36398, 36400, 36403, 36407, 37418
 - \box_gclear_new:N 312, 36404, 36406, 36409
 - \box_gframe:Nnn 322, 37168, 37174, 37179
 - \box_gresize_to_ht:Nn 320, 36984, 36987, 36989
 - \box_gresize_to_ht_plus_dp:Nn ... 320, 36984, 37007, 37009
 - \box_gresize_to_wd:Nn 321, 36984, 37027, 37029
 - \box_gresize_to_wd_and_ht:Nnn ... 321, 36984, 37044, 37046
 - \box_gresize_to_wd_and_ht_plus_dp:Nnn 321, 36935, 36941, 36946, 38062
 - \box_grotate:Nn 321, 36817, 36820, 36822, 37897
 - \box_gscale:Nnn 321, 37062, 37065, 37067, 38104
 - \box_gset_clipped:N 322, 37240, 37243, 37245
 - \box_gset_dp:Nn 313, 36435, 36441, 36443
 - \box_gset_eq:NN 312, 36401, 36410, 36412, 36415, 37250, 37301, 37595, 42225
 - \box_gset_eq_drop:NN 319, 36416, 36418, 36421, 42226
 - \box_gset_ht:Nn 313, 36435, 36450, 36452
 - \box_gset_to_last:N 314, 36489, 36491, 36494, 42227
 - \box_gset_trim:Nnnnn 322, 37246, 37249, 37251
 - \box_gset_viewport:Nnnnn 323, 37297, 37300, 37302
 - \box_gset_wd:Nn 313, 36435, 36459, 36461
 - \box_gunderline:Nnn 322, 37209, 37216, 37222
 - \box_ht:N 313, 1422, 25255, 36426, 36426, 36429, 36433, 36447, 36451, 36662, 36671, 36721, 36835, 36964, 37079, 37092, 37095, 37099, 37105, 37199, 37277, 37285, 37290, 37491, 37568, 37612, 37613, 37720, 37725, 37762, 37769, 37942, 37946, 38225, 38246, 38555
 - \box_ht_plus_dp:N 313, 36432, 36432, 36434, 37614, 37615
 - \box_if_empty:N 36485, 36487
 - \box_if_empty:NTF 313, 36485
 - \box_if_empty_p:N 313, 36485
 - \box_if_exist:N 36422, 36424
 - \box_if_exist:NTF 312, 36405, 36407, 36422, 36520
 - \box_if_exist_p:N 312, 36422
 - \box_if_horizontal:N .. 36477, 36481

- \box_if_horizontal:NTF ... [313](#), [36477](#)
- \box_if_horizontal_p:N ... [313](#), [36477](#)
- \box_if_vertical:N ... [36479](#), [36483](#)
- \box_if_vertical:NTF ... [313](#), [36477](#)
- \box_if_vertical_p:N ... [313](#), [36477](#)
- \box_log:N ... [314](#), [36506](#), [36506](#), [36508](#)
- \box_log:Nnn ... [314](#), [36506](#), [36507](#), [36509](#), [36517](#)
- \box_move_down:nn ... [312](#), [1444](#), [36466](#), [36472](#), [37187](#), [37229](#), [37264](#), [37272](#), [37315](#), [37322](#), [37921](#)
- \box_move_left:nn ... [312](#), [36466](#), [36466](#)
- \box_move_right:nn ... [312](#), [36466](#), [36468](#)
- \box_move_up:nn ... [312](#), [36466](#), [36470](#), [37281](#), [37290](#), [37329](#), [37342](#), [38266](#), [38552](#)
- \box_new:N ... [311](#), [312](#), [36392](#), [36392](#), [36397](#), [36405](#), [36407](#), [36495](#), [36496](#), [36497](#), [36498](#), [36499](#), [36816](#), [37352](#), [37425](#), [40587](#)
- \box_resize_to_ht:Nn ... [320](#), [36984](#), [36984](#), [36986](#)
- \box_resize_to_ht_plus_dp:Nn ... [320](#), [36984](#), [37004](#), [37006](#)
- \box_resize_to_wd:Nn ... [321](#), [36984](#), [37024](#), [37026](#)
- \box_resize_to_wd_and_ht:Nnn ... [321](#), [36984](#), [37041](#), [37043](#)
- \box_resize_to_wd_and_ht_plus_dp:Nnn ... [321](#), [36935](#), [36935](#), [36940](#), [38055](#)
- \box_rotate:Nn ... [321](#), [36817](#), [36817](#), [36819](#), [37894](#)
- \box_scale:Nnn ... [321](#), [37062](#), [37062](#), [37064](#), [38101](#)
- \box_set_clipped:N ... [322](#), [323](#), [37240](#), [37240](#), [37242](#)
- \box_set_dp:Nn ... [313](#), [1444](#), [36435](#), [36435](#), [36440](#), [36722](#), [36862](#), [37134](#), [37137](#), [37267](#), [37275](#), [37318](#), [37323](#), [37926](#), [38226](#), [38247](#), [38556](#), [40684](#)
- \box_set_eq:NN ... [312](#), [36399](#), [36410](#), [36410](#), [36414](#), [37247](#), [37298](#), [37583](#), [38249](#), [38560](#), [42144](#)
- \box_set_eq_drop:NN ... [319](#), [36416](#), [36416](#), [36420](#), [42145](#)
- \box_set_ht:Nn ... [313](#), [36435](#), [36444](#), [36449](#), [36720](#), [36861](#), [37133](#), [37138](#), [37284](#), [37293](#), [37332](#), [37345](#), [37924](#), [38225](#), [38246](#), [38554](#), [40685](#)
- \box_set_to_last:N ... [314](#), [36489](#), [36489](#), [36493](#), [42146](#)
- \box_set_trim:Nnnnn ... [322](#), [37246](#), [37246](#), [37248](#)
- \box_set_viewport:Nnnnn ... [323](#), [37297](#), [37297](#), [37299](#)
- \box_set_wd:Nn ... [313](#), [36435](#), [36453](#), [36458](#), [36863](#), [37150](#), [37927](#), [38227](#), [38248](#), [38558](#), [40687](#)
- \box_show:N ... [314](#), [319](#), [329](#), [36500](#), [36500](#), [36502](#)
- \box_show:Nnn ... [314](#), [330](#), [1478](#), [36500](#), [36501](#), [36503](#), [36505](#), [38594](#), [38597](#)
- \box_underline:Nnn ... [322](#), [37209](#), [37209](#), [37215](#)
- \box_use:N ... [312](#), [36462](#), [36463](#), [36465](#), [36850](#), [37185](#), [37227](#), [37257](#), [37308](#), [37922](#), [38263](#), [38266](#), [38549](#), [38552](#)
- \box_use_drop:N ... [319](#), [36462](#), [36462](#), [36464](#), [36672](#), [36724](#), [36865](#), [37145](#), [37154](#), [37265](#), [37273](#), [37282](#), [37291](#), [37316](#), [37322](#), [37330](#), [37343](#), [37929](#), [38355](#), [38484](#), [40689](#)
- \box_wd:N ... [313](#), [25254](#), [36426](#), [36428](#), [36431](#), [36456](#), [36460](#), [36837](#), [36966](#), [37081](#), [37113](#), [37186](#), [37195](#), [37228](#), [37233](#), [37309](#), [37616](#), [37617](#), [37724](#), [37733](#), [37751](#), [37756](#), [37945](#), [37953](#), [38147](#), [38154](#), [38180](#), [38227](#), [38248](#), [38264](#), [38550](#), [38559](#)
- \c_empty_box ... [311](#), [313](#), [314](#), [36399](#), [36401](#), [36495](#)
- \g_tmpa_box ... [314](#), [36496](#)
- \l_tmpa_box ... [314](#), [36496](#)
- \g_tmpb_box ... [314](#), [36496](#)
- \l_tmpb_box ... [314](#), [36496](#)
- box internal commands:
 - \l_box_angle_fp ... [36805](#), [36827](#), [36828](#), [36829](#), [36858](#)
 - _box_autosize:NnnnN ... [37091](#), [37092](#), [37095](#), [37099](#), [37105](#), [37109](#)
 - _box_backend_clip:N ... [37241](#), [37244](#)
 - _box_backend_rotate:Nn ... [36856](#)
 - _box_backend_scale:Nnn ... [37126](#)
 - \l_box_bottom_dim ... [36808](#), [36836](#), [36893](#), [36897](#), [36902](#), [36908](#), [36913](#), [36917](#), [36926](#), [36928](#), [36957](#), [36965](#), [36974](#), [37018](#), [37080](#), [37086](#)
 - \l_box_bottom_new_dim ... [36812](#), [36862](#), [36894](#), [36905](#), [36916](#), [36927](#), [36973](#), [37085](#), [37134](#), [37138](#)
 - \l_box_cos_fp ... [36806](#), [36829](#), [36841](#), [36846](#), [36873](#), [36885](#)
 - \l_box_delta_dim ... [36630](#), [36661](#), [36666](#), [36671](#)
 - _box_dim_eval:n ... [1422](#), [36389](#), [36390](#), [36433](#), [36438](#), [36442](#), [36447](#), [36451](#), [36456](#), [36460](#), [36467](#), [36469](#)

36471, 36473, 36552, 36557, 36584,
 36590, 36598, 36637, 36663, 36665,
 36670, 36680, 36689, 36752, 36757,
 36780, 36786, 36797, 36802, 37162,
 37163, 37167, 37318, 37342, 42585
 __box_dim_eval:w
 36389, 36389, 36391, 42586
 __box_frame:nnNN
 .. 37168, 37170, 37176, 37180, 37208
 \l__box_left_dim ... 36808, 36838,
 36893, 36895, 36904, 36908, 36913,
 36919, 36924, 36928, 36967, 37082
 \l__box_left_new_dim 36812, 36853,
 36864, 36896, 36907, 36918, 36929
 __box_log:nNnn . 36506, 36510, 36511
 __box_resize:N ... 36935, 36959,
 36969, 37001, 37021, 37038, 37059
 __box_resize:NNN
 .. 36935, 36971, 36973, 36975, 36979
 __box_resize_common:N
 36977, 37089, 37122, 37122
 __box_resize_set_corners:N
 36935, 36951,
 36962, 36994, 37014, 37034, 37051
 __box_resize_to_ht:NnN
 36984, 36985, 36988, 36990
 __box_resize_to_ht_plus_dp:NnN .
 36984, 37005, 37008, 37010
 __box_resize_to_wd:NnN
 36984, 37025, 37028, 37030
 __box_resize_to_wd_and_ht:NnnN .
 37042, 37045, 37047
 __box_resize_to_wd_and_ht_plus_
 dp:NnnN . 36935, 36937, 36943, 36947
 __box_resize_to_wd_ht:NnnN .. 36984
 \l__box_right_dim .. 36808, 36837,
 36891, 36897, 36902, 36906, 36915,
 36917, 36926, 36930, 36953, 36966,
 36972, 37036, 37053, 37081, 37088
 \l__box_right_new_dim ... 36812,
 36864, 36898, 36909, 36920, 36931,
 36971, 37087, 37142, 37144, 37150
 __box_rotate:N . 36817, 36830, 36833
 __box_rotate:NnN
 36817, 36818, 36821, 36823
 __box_rotate_quadrant_four: ...
 36817, 36848, 36922
 __box_rotate_quadrant_one:
 36817, 36842, 36889
 __box_rotate_quadrant_three: ...
 36817, 36847, 36911
 __box_rotate_quadrant_two:
 36817, 36843, 36900
 __box_rotate_xdir:nnN
 36817, 36867, 36895, 36897, 36906,
 36908, 36917, 36919, 36928, 36930
 __box_rotate_ydir:nnN
 36817, 36878, 36891, 36893, 36902,
 36904, 36913, 36915, 36924, 36926
 __box_rule_horizontal:nn
 .. 37159, 37159, 37191, 37203, 37234
 __box_rule_vertical:n
 37159, 37166, 37194, 37201
 __box_scale:N
 37062, 37074, 37077, 37119
 __box_scale:NnnN
 37062, 37063, 37066, 37068
 \l__box_scale_x_fp 36933,
 36952, 36972, 37000, 37020, 37035,
 37037, 37052, 37072, 37088, 37113,
 37116, 37117, 37118, 37128, 37140
 \l__box_scale_y_fp
 36933, 36954, 36974, 36976,
 36995, 37000, 37015, 37020, 37037,
 37054, 37073, 37084, 37086, 37114,
 37116, 37117, 37118, 37129, 37131
 __box_set_trim:NnnnnN
 37246, 37247, 37250, 37252
 __box_set_viewport:NnnnnN
 37298, 37301, 37303
 __box_show:NnN
 .. 36504, 36514, 36518, 36518, 36535
 \l__box_sin_fp
 .. 36806, 36828, 36839, 36874, 36884
 \l__box_tmp_box 36641,
 36656, 36662, 36663, 36670, 36671,
 36672, 36702, 36715, 36720, 36721,
 36722, 36723, 36724, 36816, 36850,
 36851, 36857, 36861, 36862, 36863,
 36865, 37124, 37133, 37134, 37137,
 37138, 37145, 37150, 37154, 37254,
 37262, 37265, 37267, 37270, 37273,
 37275, 37277, 37279, 37282, 37284,
 37285, 37288, 37290, 37291, 37293,
 37295, 37305, 37313, 37316, 37318,
 37321, 37322, 37323, 37327, 37330,
 37332, 37340, 37343, 37345, 37347
 __box_to_ht:Nnn
 36674, 36675, 36677, 36678
 __box_to_ht:Nnw
 36674, 36684, 36686, 36687
 \l__box_top_dim 36808, 36835, 36891,
 36895, 36904, 36906, 36915, 36919,
 36924, 36930, 36957, 36964, 36976,
 36998, 37018, 37057, 37079, 37084
 \l__box_top_new_dim
 36812, 36861, 36892, 36903, 36914,

- 36925, 36975, 37083, 37133, 37137
 __box_underline:nnNN
 .. 37209, 37211, 37218, 37223, 37239
 __box_vcenter:Nnnn
 .. 36632, 36649, 36651, 36653, 36654
 __box_vcenter:nnw
 .. 36632, 36633, 36635, 36637, 36638
 __box_vcenter_aux:
 36632, 36643, 36646
 __box_vcenter_calc:n
 36632, 36647, 36657, 36659
 __box_viewport:NnnnnN 37297
 __box_voffset:Nnnn
 36695, 36710, 36712, 36713
 __box_voffset:Nnnw
 36695, 36696, 36698, 36699
 __box_voffset_aux:
 36695, 36704, 36707
 __box_voffset_calc:n
 36695, 36708, 36716, 36718
 \l__box_voffset_dim
 .. 36630, 36640, 36647, 36701, 36708
 \boxdir 792
 \boxdirection 793
 \boxmaxdepth 170
 bp 287
 \breakafterdirmode 794
 \brokenpenalty 171
- C**
- \c 33333,
 35799, 35822, 35843, 35869, 35926,
 35927, 35946, 35947, 35950, 35951,
 35958, 35959, 35970, 35971, 35978,
 35979, 35982, 35983, 36038, 36039
 \catcode 66, 85, 86, 87, 88, 89, 90, 91, 92,
 96, 97, 98, 99, 100, 101, 102, 103, 172
 \catcodetable 795
 cc 287
 cctab commands:
 \cctab_begin:N .. 297, 1306, 1307,
 1309, 1311–1314, 31662, 31662, 31675
 \cctab_const:Nn
 ... 296, 297, 1307, 31795, 31795,
 31805, 31807, 31814, 31856, 42304
 \cctab_end: 297,
 1306, 1307, 1310–1314, 31676, 31676
 \cctab_gsave_current:N
 296, 31598, 31598, 31603
 \cctab_gset:Nn 296,
 297, 1316, 31586, 31586, 31597, 42228
 \cctab_if_exist:N 31750, 31752
 \cctab_if_exist:NTF 297, 31750, 31757
 \cctab_if_exist_p:N 297, 31750
 \cctab_item:Nn 297, 31734, 31734, 31749
 \cctab_new:N 296, 1306,
 1307, 1316, 31517, 31519, 31539,
 31558, 31806, 31867, 31868, 42334
 \cctab_select:N ... 129, 130, 296,
 297, 14706, 31591, 31614, 31614,
 31616, 31800, 31809, 31816, 31858
 \c_code_cctab 297, 14706, 31819
 \c_document_cctab ... 297, 1309, 31819
 \c_initex_cctab
 298, 31591, 31800, 31806
 \c_other_cctab 298, 31806
 \g_tmpa_cctab 298, 31867
 \g_tmpb_cctab 298, 31867
 cctab internal commands:
 \g__cctab_allocate_int
 31513, 31655, 31657, 31659
 __cctab_begin_aux:
 ... 1311, 31643, 31645, 31653, 31667
 __cctab_chk_group_begin:n
 ... 1312, 31668, 31687, 31687, 31693
 __cctab_chk_group_end:n
 1312, 31681, 31687, 31694
 __cctab_chk_if_valid:N 31754
 __cctab_chk_if_valid:NTF
 .. 31588, 31600, 31615, 31664, 31754
 __cctab_chk_if_valid_aux:NTF ...
 .. 31754, 31759, 31775, 31781, 31788
 \g__cctab_endlinechar_prop
 ... 1308, 31516, 31567, 31569, 31622
 \g__cctab_group_seq
 31512, 31689, 31696
 __cctab_gset:n 31559, 31561, 31575,
 31593, 31601, 31671, 31802, 31854
 __cctab_gset_aux:n
 31559, 31562, 31563
 __cctab_gstore:Nnn
 31517, 31537, 31546, 31547, 31548,
 31549, 31551, 31552, 31554, 31555
 __cctab_item:nN . 31735, 31738, 31742
 __cctab_nesting_number:N
 .. 31669, 31682, 31712, 31713, 31715
 __cctab_nesting_number:w
 31712, 31717, 31722
 __cctab_new:N . 1307, 1311, 31517,
 31522, 31524, 31531, 31542, 31606,
 31626, 31647, 31656, 31798, 31823
 \g__cctab_next_cctab 31643
 __cctab_select:N ... 1310, 31614,
 31615, 31619, 31632, 31672, 31683
 \g__cctab_stack_seq
 ... 1306, 31510, 31670, 31678, 31730
 \g__cctab_tmp_cctab 31604

- _cctab_tmp_cctab_name: . [31604](#),
[31607](#), [31625](#), [31626](#), [31627](#), [31628](#)
- \l_cctab_tmpa_tl
. [1311](#), [1312](#), [31514](#), [31622](#), [31623](#),
[31648](#), [31658](#), [31666](#), [31669](#), [31670](#),
[31671](#), [31678](#), [31680](#), [31682](#), [31683](#)
- \l_cctab_tmpp_tl
..... [31514](#), [31696](#), [31700](#), [31707](#)
- \g_cctab_unused_seq
[1306](#), [1311](#), [1312](#), [31510](#), [31666](#), [31680](#)
- ceil [283](#)
- \char [173](#), [20322](#)
- char commands:
- \l_char_active_seq ... [93](#), [205](#), [19962](#)
- \char_fold_case:N [41605](#), [41612](#)
- \char_foldcase:N [41621](#), [41628](#)
- \char_generate:nn [129](#), [202](#),
[464](#), [485](#), [486](#), [572](#), [722](#), [790](#), [936](#),
[3612](#), [3613](#), [3614](#), [3615](#), [3617](#), [3618](#),
[3619](#), [4194](#), [4268](#), [4284](#), [4296](#), [4717](#),
[5755](#), [6129](#), [12519](#), [12535](#), [12580](#),
[12614](#), [12618](#), [12632](#), [14547](#), [14804](#),
[14820](#), [19990](#), [19990](#), [20089](#), [20792](#),
[31921](#), [31929](#), [31948](#), [31951](#), [31954](#),
[31956](#), [31968](#), [31999](#), [32718](#), [32762](#),
[34618](#), [34628](#), [34776](#), [34798](#), [39325](#)
- \char_gset_active_eq:NN
..... [202](#), [19968](#), [19985](#)
- \char_gset_active_eq:nN
..... [202](#), [19968](#), [19987](#)
- \char_lower_case:N [41605](#), [41606](#)
- \char_lowercase:N [41621](#), [41622](#)
- \char_mixed_case:N [41610](#)
- \char_mixed_case:Nn [41605](#)
- \char_set_active_eq:NN
..... [202](#), [3609](#), [4190](#), [19968](#), [19984](#)
- \char_set_active_eq:nN
..... [202](#), [4231](#), [4232](#), [19968](#), [19986](#)
- \char_set_catcode:nn .. [204](#), [112](#),
[113](#), [114](#), [115](#), [116](#), [117](#), [118](#), [119](#),
[19868](#), [19868](#), [19875](#), [19877](#), [19879](#),
[19881](#), [19883](#), [19885](#), [19887](#), [19889](#),
[19891](#), [19893](#), [19895](#), [19897](#), [19899](#),
[19901](#), [19903](#), [19905](#), [19907](#), [19909](#),
[19911](#), [19913](#), [19915](#), [19917](#), [19919](#),
[19921](#), [19923](#), [19925](#), [19927](#), [19929](#),
[19931](#), [19933](#), [19935](#), [19937](#), [31636](#)
- \char_set_catcode_active:N
..... [203](#), [3669](#),
[3689](#), [7251](#), [9448](#), [19874](#), [19900](#),
[19969](#), [20023](#), [20085](#), [20160](#), [35740](#)
- \char_set_catcode_active:n
..... [203](#), [9197](#),
[19906](#), [19932](#), [20033](#), [22348](#), [22349](#),
[31829](#), [31836](#), [31853](#), [31864](#), [32686](#)
- \char_set_catcode_alignment:N ...
..... [203](#), [7303](#), [19874](#), [19882](#), [20148](#)
- \char_set_catcode_alignment:n ...
.... [203](#), [19906](#), [19914](#), [20048](#), [31842](#)
- \char_set_catcode_comment:N
..... [203](#), [19874](#), [19902](#)
- \char_set_catcode_comment:n
..... [203](#), [19906](#), [19934](#), [31841](#)
- \char_set_catcode_end_line:N ...
..... [203](#), [19874](#), [19884](#)
- \char_set_catcode_end_line:n ...
..... [203](#), [19906](#), [19916](#), [31837](#)
- \char_set_catcode_escape:N
..... [203](#), [19874](#), [19874](#)
- \char_set_catcode_escape:n
..... [203](#), [19906](#), [19906](#), [31844](#)
- \char_set_catcode_group_begin:N .
..... [203](#), [3766](#), [7254](#), [19874](#), [19876](#)
- \char_set_catcode_group_begin:n .
.... [203](#), [19906](#), [19908](#), [20054](#), [31847](#)
- \char_set_catcode_group_end:N ...
..... [203](#), [3769](#), [7271](#), [19874](#), [19878](#)
- \char_set_catcode_group_end:n ...
.... [203](#), [19906](#), [19910](#), [20052](#), [31849](#)
- \char_set_catcode_ignore:N
..... [203](#), [19874](#), [19892](#)
- \char_set_catcode_ignore:n
..... [203](#), [126](#),
[127](#), [19906](#), [19924](#), [31834](#), [31838](#), [32460](#)
- \char_set_catcode_invalid:N
..... [203](#), [19874](#), [19904](#)
- \char_set_catcode_invalid:n
..... [203](#), [19906](#), [19936](#), [31826](#), [31828](#), [31851](#)
- \char_set_catcode_letter:N
..... [203](#), [7280](#), [19874](#), [19896](#), [26945](#), [26946](#)
- \char_set_catcode_letter:n
.... [203](#), [129](#), [131](#), [19906](#), [19928](#),
[20037](#), [31831](#), [31833](#), [31843](#), [31846](#)
- \char_set_catcode_math_subscript:N
..... [203](#), [7268](#), [19874](#), [19890](#), [20152](#)
- \char_set_catcode_math_subscript:n
.... [203](#), [19906](#), [19922](#), [20041](#), [31863](#)
- \char_set_catcode_math_superscript:N
..... [203](#), [7306](#), [19874](#), [19888](#)
- \char_set_catcode_math_superscript:n
. [203](#), [130](#), [19906](#), [19920](#), [20043](#), [31845](#)
- \char_set_catcode_math_toggle:N .
..... [203](#), [7283](#), [19874](#), [19880](#), [20146](#)
- \char_set_catcode_math_toggle:n .
.... [203](#), [19906](#), [19912](#), [20050](#), [31840](#)
- \char_set_catcode_other:N .. [203](#),
[1309](#), [3932](#), [7286](#), [15095](#), [15096](#),

- 15432, 15433, 15614, 15615, 15616,
19874, 19898, 42005, 42067, 42493
- \char_set_catcode_other:n
..... 203, 128, 132, 9192, 9194,
9196, 19906, 19930, 20035, 31812,
31830, 31832, 31835, 31848, 31862
- \char_set_catcode_parameter:N ...
..... 203, 7289,
19874, 19886, 42010, 42072, 42498
- \char_set_catcode_parameter:n ...
..... 203, 19906, 19918, 20045, 31839
- \char_set_catcode_space:N
..... 203, 19874, 19894, 42011, 42073
- \char_set_catcode_space:n
..... 203, 133,
11668, 19906, 19926, 31817, 31850,
31860, 31861, 32452, 40570, 40571
- \char_set_lccode:nn
204, 9444, 9445, 9446, 9447, 19938,
19944, 19975, 20058, 20059, 20086
- \char_set_mathcode:nn
..... 205, 19938, 19938
- \char_set_sfcode:nn 205, 19938, 19956
- \char_set_uccode:nn 204, 19938, 19950
- \char_show_value_catcode:n
..... 204, 19868, 19872
- \char_show_value_lccode:n
..... 204, 19938, 19948
- \char_show_value_mathcode:n ...
..... 205, 19938, 19942
- \char_show_value_sfcode:n
..... 205, 19938, 19960
- \char_show_value_uccode:n
..... 205, 19938, 19954
- \l_char_special_seq 205, 19962
- \char_str_fold_case:N . 41605, 41620
- \char_str_foldcase:N .. 41621, 41638
- \char_str_lower_case:N . 41605, 41614
- \char_str_lowercase:N . 41621, 41630
- \char_str_mixed_case:N . 41605, 41618
- \char_str_titlecase:N . 41621, 41633
- \char_str_upper_case:N . 41605, 41616
- \char_str_uppercase:N . 41621, 41636
- \char_titlecase:N 41621, 41626
- \char_to_nfd:N 41601, 41602
- \char_to_nfd:n 41601, 41604
- \char_to_utfviii_bytes:n 41599, 41600
- \char_upper_case:N 41605, 41608
- \char_uppercase:N 41621, 41624
- \char_value_catcode:n
..... 204, 1314, 112, 113,
114, 115, 116, 117, 118, 119, 12531,
12535, 12577, 12580, 19868, 19870,
19873, 31580, 31746, 32101, 32108,
32720, 33859, 33863, 33880, 33946,
33988, 34179, 35755, 35806, 35833
- \char_value_lccode:n
..... 204, 19938, 19946, 19949
- \char_value_mathcode:n
..... 205, 19938, 19940, 19943
- \char_value_sfcode:n
..... 205, 19938, 19958, 19961
- \char_value_uccode:n
..... 205, 19938, 19952, 19955
- char internal commands:
- __char_generate_aux:nn 19990
- __char_generate_aux:nnw
..... 19990, 20015, 20026, 20068
- __char_generate_aux:w . 19992, 19996
- __char_generate_auxii:nnw ... 19990
- __char_generate_invalid_-
catcode: 19990
- __char_int_to_roman:w
..... 19988, 19988, 20063, 20078
- __char_quark_if_no_value:N .. 19867
- __char_quark_if_no_value:NTF . 19867
- __char_quark_if_no_value_p:N . 19867
- __char_sep:
.. 19989, 19989, 19993, 19994, 19996
- __char_tmp:n 20056, 20067
- __char_tmp:nN .. 19970, 19981, 19982
- \l_char_tmp_tl 19990
- \chardef 1315, 94, 105, 174
- choice commands:
- .choice: 247, 22958
- choices commands:
- .choices:nn 247, 22960
- \cite 1344, 32936, 32946
- \cleaders 175
- \clearmarks 796
- clist commands:
- \clist_clear:N
..... 191, 19239, 19239, 19240,
19256, 19413, 23138, 32408, 42147
- \clist_clear_new:N
..... 191, 19243, 19243, 19244
- \clist_concat:NNN
..... 192, 1568, 1570, 19282,
19282, 19295, 19310, 19323, 42123
- \clist_const:Nn
.... 191, 19235, 19235, 19237, 19238
- \clist_count:N 196, 199,
19658, 19658, 19666, 19692, 19759,
19826, 19837, 32303, 32349, 32358
- \clist_count:n . 196, 19658, 19670,
19687, 19790, 19817, 19838, 39847
- \clist_gclear:N
191, 19239, 19241, 19242, 19258, 42229

- \clist_gclear_new:N
..... [191](#), [19243](#), [19245](#), [19246](#)
- \clist_gconcat:NNN ... [192](#), [19282](#),
[19284](#), [19296](#), [19312](#), [19325](#), [42124](#)
- \clist_get:NN
[198](#), [19339](#), [19339](#), [19349](#), [19376](#), [19385](#)
- \clist_get:NNTF [198](#), [19376](#)
- \clist_gpop:NN
[198](#), [19350](#), [19352](#), [19375](#), [19388](#), [19400](#)
- \clist_gpop:NNTF [198](#), [19376](#)
- \clist_gpush:Nn
..... [198](#), [19401](#), [19403](#), [19404](#)
- \clist_gput_left:Nn
[192](#), [19309](#), [19311](#), [19320](#), [19321](#), [19403](#)
- \clist_gput_right:Nn
.... [192](#), [19322](#), [19324](#), [19335](#), [19337](#)
- \clist_gremove_all:Nn
..... [193](#), [19429](#), [19431](#), [19467](#)
- \clist_gremove_duplicates:N
..... [193](#), [19407](#), [19409](#), [19428](#)
- \clist_greverse:N
..... [193](#), [19468](#), [19470](#), [19473](#)
- \clist_gset:N [247](#), [22972](#)
- \clist_gset:Nn
[192](#), [14833](#), [19301](#), [19303](#), [19307](#), [19308](#)
- \clist_gset_eq:NN
..... [191](#), [19247](#), [19251](#), [19252](#),
[19253](#), [19254](#), [19410](#), [42104](#), [42230](#)
- \clist_gset_from_seq:NN [191](#), [3324](#),
[19255](#), [19257](#), [19280](#), [19281](#), [19432](#)
- \clist_gsort:Nn
..... [193](#), [3309](#), [3321](#), [3326](#), [19486](#)
- \clist_if_empty:N [19486](#), [19488](#)
- \clist_if_empty:n [19490](#)
- \clist_if_empty:NTF
.. [194](#), [19291](#), [19420](#), [19453](#), [19486](#),
[19543](#), [19583](#), [19615](#), [19825](#), [22736](#)
- \clist_if_empty:nTF [194](#), [19490](#)
- \clist_if_empty_p:N [194](#), [19486](#)
- \clist_if_empty_p:n [194](#), [19490](#)
- \clist_if_exist:N [19297](#), [19299](#)
- \clist_if_exist:NTF
[192](#), [11520](#), [11636](#), [19297](#), [19690](#), [40643](#)
- \clist_if_exist_p:N [192](#), [19297](#)
- \clist_if_in:Nn [19504](#), [19537](#)
- \clist_if_in:nn [19508](#), [19539](#)
- \clist_if_in:NnTF
.. [191](#), [194](#), [1039](#), [19416](#), [19504](#), [23349](#)
- \clist_if_in:nnTF .. [194](#), [19504](#), [24622](#)
- \clist_item:Nn
[199](#), [927](#), [19756](#), [19756](#), [19786](#), [19826](#)
- \clist_item:nn
[199](#), [927](#), [19787](#), [19787](#), [19795](#), [19821](#)
- \clist_log:N . [199](#), [19829](#), [19831](#), [19832](#)
- \clist_log:n [199](#), [19851](#), [19852](#)
- \clist_map_break: [195](#), [19548](#),
[19560](#), [19569](#), [19570](#), [19593](#), [19620](#),
[19633](#), [19641](#), [19642](#), [19654](#), [19654](#),
[19655](#), [19657](#), [23352](#), [23404](#), [23453](#)
- \clist_map_break:n [196](#), [3317](#), [3323](#),
[19524](#), [19654](#), [19656](#), [23443](#), [39824](#)
- \clist_map_function:NN
.. [194](#), [922](#), [17412](#), [17422](#), [19527](#),
[19541](#), [19541](#), [19564](#), [19663](#), [19842](#)
- \clist_map_function:nN
..... [194](#), [195](#), [923](#), [14836](#),
[17417](#), [17427](#), [17438](#), [19565](#), [19565](#),
[19572](#), [19856](#), [23566](#), [39985](#), [40061](#)
- \clist_map_inline:Nn
..... [194](#), [195](#), [3317](#), [3323](#),
[9283](#), [19414](#), [19581](#), [19581](#), [19600](#),
[19602](#), [23347](#), [23395](#), [23429](#), [23450](#)
- \clist_map_inline:nn
..... [195](#), [3108](#), [10247](#),
[11824](#), [11855](#), [11867](#), [19581](#), [19597](#),
[21690](#), [22690](#), [22817](#), [23945](#), [24129](#),
[31042](#), [32132](#), [32412](#), [39401](#), [39604](#),
[39606](#), [39642](#), [39811](#), [39965](#), [40009](#),
[40014](#), [41689](#), [41697](#), [42654](#), [42678](#)
- \clist_map_tokens:Nn
[195](#), [922](#), [19604](#), [19613](#), [19613](#), [19637](#)
- \clist_map_tokens:nn [195](#), [19638](#), [19638](#)
- \clist_map_variable:NnNn
.... [195](#), [19603](#), [19603](#), [19605](#), [19611](#)
- \clist_map_variable:nNn
..... [195](#), [19603](#), [19608](#)
- \clist_new:N
.. [191](#), [909](#), [19233](#), [19233](#), [19234](#),
[19405](#), [19858](#), [19859](#), [19860](#), [19861](#),
[22461](#), [22463](#), [22477](#), [22478](#), [22479](#)
- \clist_pop:NN
[198](#), [19350](#), [19350](#), [19374](#), [19386](#), [19399](#)
- \clist_pop:NNTF [198](#), [19376](#)
- \clist_push:Nn [198](#), [19401](#), [19401](#), [19402](#)
- \clist_put_left:Nn
[192](#), [19309](#), [19309](#), [19318](#), [19319](#), [19401](#)
- \clist_put_right:Nn .. [192](#), [19322](#),
[19322](#), [19331](#), [19333](#), [22913](#), [23497](#),
[23507](#), [23535](#), [32302](#), [32357](#), [32374](#)
- \clist_rand_item:N
..... [199](#), [19816](#), [19823](#), [19828](#)
- \clist_rand_item:n
..... [79](#), [199](#), [19816](#), [19816](#)
- \clist_remove_all:Nn
[193](#), [9298](#), [19429](#), [19429](#), [19466](#), [22914](#)
- \clist_remove_duplicates:N
..... [191](#), [193](#), [19407](#), [19407](#), [19427](#)

- \clist_reverse:N
..... [193](#), [19468](#), [19468](#), [19472](#)
- \clist_reverse:n
..... [193](#), [918](#), [19469](#), [19471](#), [19474](#), [19474](#)
- .clist_set:N [247](#), [22972](#)
- \clist_set:Nn [192](#),
[197](#), [19301](#), [19301](#), [19305](#), [19306](#),
[19310](#), [19312](#), [19323](#), [19325](#), [19510](#),
[19599](#), [19610](#), [22735](#), [22749](#), [23139](#)
- \clist_set_eq:NN [191](#), [19247](#),
[19247](#), [19248](#), [19249](#), [19250](#), [19408](#),
[23143](#), [23334](#), [32388](#), [42103](#), [42148](#)
- \clist_set_from_seq:NN . [191](#), [3318](#),
[19255](#), [19255](#), [19278](#), [19279](#), [19430](#)
- \clist_show:N [199](#), [19829](#), [19829](#), [19830](#)
- \clist_show:n [199](#), [19851](#), [19851](#)
- \clist_sort:Nn
..... [193](#), [3309](#), [3315](#), [3320](#), [19486](#)
- \clist_use:N . [197](#), [19721](#), [19721](#), [19722](#)
- \clist_use:Nn [197](#), [19688](#), [19718](#), [19720](#)
- \clist_use:nn [197](#), [19723](#), [19755](#)
- \clist_use:Nnnn [196](#),
[197](#), [877](#), [19688](#), [19688](#), [19711](#), [19719](#)
- \clist_use:nnnn
..... [197](#), [19723](#), [19723](#), [19755](#)
- \clist_use:Nnnnn [197](#)
- \c_empty_clist
..... [200](#), [19180](#), [19341](#), [19356](#), [19378](#), [19392](#)
- \g_tmpa_clist [200](#), [19858](#)
- \l_tmpa_clist [200](#), [19858](#)
- \g_tmpb_clist [200](#), [19858](#)
- \l_tmpb_clist [200](#), [19858](#)
- clist internal commands:
- __clist_concat:NNNN
..... [19282](#), [19283](#), [19285](#), [19286](#)
- __clist_count:n . [19658](#), [19663](#), [19667](#)
- __clist_count:w
..... [19658](#), [19675](#), [19679](#), [19683](#)
- __clist_get:wN
..... [19339](#), [19344](#), [19347](#), [19381](#)
- __clist_if_empty_n:w
..... [19490](#), [19492](#), [19497](#), [19500](#)
- __clist_if_empty_n:wNw
..... [19490](#), [19501](#), [19503](#)
- __clist_if_in_return:nnN
..... [19504](#), [19506](#), [19511](#), [19514](#)
- __clist_if_wrap:n [19207](#)
- __clist_if_wrap:nTF . [910](#), [19207](#),
[19232](#), [19274](#), [19421](#), [19435](#), [19516](#)
- __clist_if_wrap:w
..... [910](#), [19207](#), [19211](#), [19230](#)
- __clist_item:nnnN
.. [19756](#), [19758](#), [19764](#), [19779](#), [19789](#)
- __clist_item_n:nw [19787](#), [19793](#), [19796](#)
- __clist_item_n_end:n
..... [19787](#), [19804](#), [19812](#)
- __clist_item_N_loop:nw
..... [19756](#), [19762](#), [19780](#), [19784](#)
- __clist_item_n_loop:nw
.. [19787](#), [19797](#), [19798](#), [19801](#), [19806](#)
- __clist_item_n_strip:n
..... [19787](#), [19813](#), [19814](#)
- __clist_item_n_strip:w
..... [19787](#), [19814](#), [19815](#)
- __clist_map_function:Nw
[920](#), [19541](#), [19545](#), [19551](#), [19556](#), [19588](#)
- __clist_map_function_end:w
.... [920](#), [19541](#), [19554](#), [19558](#), [19562](#)
- __clist_map_function_n:Nn
.... [921](#), [19565](#), [19567](#), [19573](#), [19577](#)
- __clist_map_tokens:nw
..... [19613](#), [19617](#), [19623](#), [19629](#)
- __clist_map_tokens_end:w
..... [19613](#), [19626](#), [19631](#), [19635](#)
- __clist_map_tokens_n:nw
..... [19638](#), [19640](#), [19644](#), [19652](#)
- __clist_map_unbrace:wn
[921](#), [19565](#), [19576](#), [19580](#), [19650](#), [19740](#)
- __clist_map_variable:Nnn
..... [922](#), [19603](#), [19604](#), [19606](#)
- __clist_pop:NNN
..... [19350](#), [19351](#), [19353](#), [19354](#)
- __clist_pop:wN . [19350](#), [19367](#), [19373](#)
- __clist_pop:wwNNN
.... [915](#), [19350](#), [19359](#), [19362](#), [19395](#)
- __clist_pop_TF:NNN
..... [19376](#), [19387](#), [19389](#), [19390](#)
- __clist_put_left:NNNn
..... [19309](#), [19310](#), [19312](#), [19313](#)
- __clist_put_right:NNNn
..... [19322](#), [19323](#), [19325](#), [19326](#)
- __clist_rand_item:nn
..... [19816](#), [19817](#), [19818](#)
- __clist_remove_all:
..... [19429](#), [19446](#), [19450](#), [19463](#)
- __clist_remove_all:NNNn
..... [19429](#), [19430](#), [19432](#), [19433](#)
- __clist_remove_all:w
..... [917](#), [19429](#), [19464](#), [19465](#)
- \l__clist_remove_clist ... [19405](#),
[19413](#), [19416](#), [19418](#), [19420](#), [19425](#)
- __clist_remove_duplicates:NN ...
..... [19407](#), [19408](#), [19410](#), [19411](#)
- \l__clist_remove_seq
..... [19405](#), [19437](#), [19438](#), [19439](#)
- __clist_reverse:wwNww
.... [918](#), [19474](#), [19476](#), [19477](#), [19481](#)

- __clist_reverse_end:ww 918, 19474, 19478, 19484
- __clist_sanitize:n 19194, 19194, 19236, 19302, 19304
- __clist_sanitize:Nn 910, 19194, 19196, 19200, 19204
- __clist_set_from_seq:n 19255, 19267, 19271
- __clist_set_from_seq:NNNN 19255, 19256, 19258, 19259
- __clist_show:NN 19829, 19829, 19831, 19833
- __clist_show:Nn 19851, 19851, 19852, 19853
- __clist_tmp:w 917, 19187, 19187, 19442, 19464, 19518, 19527, 19531, 19533, 19668, 19686
- \l__clist_tmp_clist 913, 19181, 19315, 19316, 19328, 19329, 19510, 19511, 19512, 19599, 19600, 19610, 19611
- __clist_trim_next:w 910, 921, 19188, 19188, 19191, 19197, 19205, 19568, 19578
- __clist_use:Nw 925, 19723, 19725, 19726, 19727, 19733, 19736, 19752
- __clist_use:nwn 19688, 19702, 19716
- __clist_use:nwnwnwn 924, 19688, 19699, 19701, 19713
- __clist_use:wn 19688, 19695, 19696, 19712
- __clist_use_end:w 925, 19723, 19727, 19746, 19752
- __clist_use_i_delimit_by_s_-stop:nw 19184, 19186, 19783
- __clist_use_more:w 925, 19723, 19728, 19749, 19752
- __clist_use_none_delimit_by_s_-mark:w 19184, 19184, 19738
- __clist_use_none_delimit_by_s_-stop:w 917, 19184, 19185, 19202, 19445, 19553, 19560, 19575, 19625, 19633, 19648, 19681, 19725, 19769, 19774
- __clist_use_one:w 19723, 19726, 19744
- __clist_wrap_item:w 910, 19203, 19231, 19231
- \closein 176
- \closeout 177
- \clubpenalties 475
- \clubpenalty 178
- cm 287
- code commands:
 - .code:n 248, 22970
- codepoint commands:
 - \codepoint_generate:nn 301, 31915, 31923, 31960, 32120, 33765, 33779, 33780, 33854, 33858, 33862, 33946, 33951, 33987, 34118, 34122, 34178, 34501, 34588, 34590, 34601, 34603, 34663, 34665, 34667, 34677, 34711, 34734, 34810, 34822, 34843, 34848, 34857, 35754, 35806, 35833, 41599
 - \codepoint_str_generate:n .. 301, 14405, 14408, 14410, 31915, 31917, 31934, 32245, 32285, 32499, 32523, 32551, 32575, 32651, 33907, 33923
 - \codepoint_to_category:n 302, 1335, 32080, 32082, 32088, 33789
 - \codepoint_to_nfd:n 302, 32098, 32098, 33978, 34513, 41601, 41602, 41603, 41604
- codepoint internal commands:
 - __codepoint_add:nn 32223, 32228, 32279, 32280, 32281, 32300
 - \c__codepoint_block_size_int ... 32127, 32140, 32304, 32348, 32359, 32362, 32367, 32370, 32373, 32424, 32438, 32470, 32475, 32487
 - __codepoint_case:nn 32572, 32586, 32587, 32590, 32593, 32595, 32598, 32606
 - __codepoint_case:nnn 32572, 32574, 32577
 - __codepoint_casefold:n 32572, 32595
 - \l__codepoint_category_Cn_tl . 32162
 - \l__codepoint_category_default_-tl 32162, 32312
 - \l__codepoint_category_next_tl .. 32274, 32295, 32297, 32298
 - __codepoint_data:nnn 32464, 32466, 32484
 - __codepoint_data_auxi:w 32145, 32150, 32152, 32163, 32175, 32189, 32209, 32232, 32458, 32492, 32533, 32538, 32568
 - __codepoint_data_auxii:w 32190, 32191, 32238, 32242, 32504, 32521, 32541, 32542, 32544, 32546
 - __codepoint_data_auxiii:w 32192, 32193, 32240, 32251
 - __codepoint_data_auxiv:w 32256, 32272
 - __codepoint_data_auxv:nnnw ... 32276, 32307
 - __codepoint_data_auxvi:nn 32288, 32289, 32293, 32322

```

\__codepoint_data_auxvi:w .....
..... 32201, 32214, 32215
\__codepoint_data_category:n ...
..... 32258, 32264
\g__codepoint_data_ior .....
..... 32128, 32203,
32204, 32212, 32451, 32453, 32491,
32530, 32536, 32537, 32559, 32570
\__codepoint_data_offset:nn ....
..... 32259, 32260, 32266, 32282
\__codepoint_data_property:nnnn .
..... 32216, 32327
\__codepoint_finalize_blocks:n ..
..... 32410, 32461
\__codepoint_finalize_blocks:nnn
..... 32427, 32435
\__codepoint_finalize_blocks:nnnw
..... 32437, 32442, 32448
\__codepoint_finalize_blocks-
aux:n ..... 32416, 32419
\__codepoint_generate:n .. 31915,
31985, 31986, 31989, 31991, 31996
\__codepoint_generate:nnnn .....
..... 31915, 31973, 31979
\l__codepoint_grapheme_default_-
tl ..... 32174
\l__codepoint_grapheme_Other_tl .
..... 32174
\__codepoint_lowercase:n 32572, 32587
\__codepoint_lowercase_base_-
mapping:n .... 32496, 32510, 32516
\l__codepoint_lowercase_default_-
tl ..... 32144
\l__codepoint_lowercase_next_tl .
..... 32297
\l__codepoint_matched_block_tl ..
.. 32142, 32378, 32383, 32386, 32404
\__codepoint_nfd:n .....
..... 32114, 32644, 32646, 32650
\__codepoint_nfd:nn .....
..... 32644, 32651, 32652
\__codepoint_nfd:w .... 12175, 32647
\__codepoint_range:nnn .....
..... 32220, 32227,
32311, 32313, 32314, 32317, 32318,
32319, 32333, 32339, 32340, 32414
\__codepoint_range:nnnn 32343, 32354
\__codepoint_range_aux:nnn .....
..... 32335, 32341
\__codepoint_save_blocks:nn ....
..... 32305, 32360, 32369, 32376
\__codepoint_str_generate:nnnn ..
..... 31915, 31941, 31946
\__codepoint_titlecase:n .....
..... 32572, 32590, 32593
\l__codepoint_tmpa_tl .....
..... 32453, 32455, 32458
\__codepoint_to_bytes_auxi:n ...
..... 32002, 32004, 32007
\__codepoint_to_bytes_auxii:Nnn .
.. 32002, 32012, 32018, 32029, 32051
\__codepoint_to_bytes_auxiii:n ..
..... 32002, 32014, 32021,
32025, 32034, 32039, 32043, 32053
\__codepoint_to_bytes_end: .....
..... 32002, 32049, 32056,
32059, 32062, 32068, 32076, 32079
\__codepoint_to_bytes_output:nnn
..... 32002, 32057,
32060, 32064, 32070, 32073, 32078
\__codepoint_to_bytes_outputi:nw
..... 32002,
32011, 32017, 32027, 32047, 32055
\__codepoint_to_bytes_outputii:nw
.. 32002, 32013, 32019, 32032, 32058
\__codepoint_to_bytes_outputiii:nw
..... 32002, 32024, 32037, 32061
\__codepoint_to_bytes_outputiv:nw
..... 32002, 32042, 32067
\__codepoint_to_nfd:n .....
..... 32098, 32099, 32100, 32104
\__codepoint_to_nfd:nn .....
..... 32098, 32101,
32107, 32108, 32111, 32122, 32124
\__codepoint_to_nfd:nnn .....
..... 32098, 32113, 32116
\__codepoint_to_nfd:nnnn .....
..... 32098, 32116, 32117
\__codepoint_uppercase:n 32572, 32586
\l__codepoint_uppercase_default_-
tl ..... 32143
\l__codepoint_uppercase_next_tl .
..... 32298
\l__codepoint_wordbreak_default_-
tl ..... 32188
\l__codepoint_wordbreak_Other_tl
..... 32188
coffin commands:
\coffin_attach:NnnNnnnn .....
..... 328, 1477, 38209, 38209, 38214
\coffin_clear:N .....
..... 325, 37405, 37405, 37413
\coffin_display_handles:Nn .....
..... 329, 38461, 38461, 38537
\coffin_dp:N ..... 328, 37610,
37610, 37611, 38073, 38112, 38576

```

- \coffin_gattach:NnnNnnnn
..... 328, 38209, 38215, 38220
- \coffin_gclear:N
..... 325, 37405, 37414, 37422
- \coffin_gjoin:NnnNnnnn
..... 328, 38158, 38164, 38169
- \coffin_greset_poles:N 327, 37458,
37471, 37530, 37548, 37694, 37700
- \coffin_gresize:Nnn
..... 327, 38052, 38059, 38065
- \coffin_grotate:Nn
..... 327, 37893, 37896, 37898
- \coffin_gscale:Nnn
..... 327, 38100, 38103, 38105
- \coffin_gset_eq:NN
325, 37579, 37591, 37602, 38167, 38218
- \coffin_gset_horizontal_pole:Nnn
..... 326, 37648, 37651, 37653
- \coffin_gset_vertical_pole:Nnn ..
..... 326, 37648, 37669, 37671
- \coffin_ht:N 328, 37610,
37612, 37613, 38073, 38112, 38575
- \coffin_ht_plus_dp:N
..... 329, 37610, 37614, 37615
- \coffin_if_exist:N 37384, 37394
- \coffin_if_exist:NTF 325, 37384, 37398
- \coffin_if_exist_p:N 325, 37384
- \coffin_join:NnnNnnnn
..... 328, 38158, 38158, 38163
- \coffin_log:N 329, 38587, 38590, 38592
- \coffin_log:Nnn
.... 330, 38587, 38591, 38596, 38598
- \coffin_log_structure:N
..... 329, 38562, 38565, 38567
- \coffin_mark_handle:Nnnn
..... 329, 38416, 38416, 38460
- \coffin_new:N 325,
1453, 37423, 37423, 37435, 37603,
37604, 37605, 37606, 37607, 37608,
37609, 38348, 38358, 38359, 38360
- \coffin_pole:Nn 327, 37618, 37618,
37787, 37788, 38312, 38313, 38319,
38320, 38471, 38472, 38475, 38476
- \coffin_reset_poles:N 327, 37445,
37465, 37517, 37541, 37694, 37694
- \coffin_resize:Nnn
..... 327, 38052, 38052, 38058
- \coffin_rotate:Nn
..... 327, 37893, 37893, 37895
- \coffin_scale:Nnn
..... 327, 38100, 38100, 38102
- \coffin_set_eq:NN 325, 37579, 37579,
37590, 38161, 38212, 38268, 38478
- \coffin_set_horizontal_pole:Nnn .
..... 326, 37648, 37648, 37650
- \coffin_set_vertical_pole:Nnn ...
..... 326, 37648, 37666, 37668
- \coffin_show:N 329, 38587, 38587, 38589
- \coffin_show:Nnn
.... 330, 38587, 38588, 38593, 38595
- \coffin_show_structure:N ... 327,
329, 330, 1478, 38562, 38562, 38564
- \coffin_typeset:Nnnnn
..... 328, 38350, 38350, 38357
- \coffin_wd:N 329, 37610,
37616, 37617, 38069, 38116, 38577
- \c_empty_coffin 330, 37603
- \g_tmpa_coffin 330, 37606
- \l_tmpa_coffin 330, 37606
- \g_tmpb_coffin 330, 37606
- \l_tmpb_coffin 330, 37606
- coffin internal commands:
- __coffin_align:NnnNnnnnN 38172,
38223, 38244, 38251, 38251, 38353
- \l_coffin_aligned_coffin
..... 37603, 38173,
38174, 38178, 38184, 38187, 38190,
38206, 38207, 38224, 38225, 38226,
38227, 38228, 38231, 38235, 38239,
38240, 38245, 38246, 38247, 38248,
38249, 38282, 38298, 38354, 38355,
38547, 38554, 38556, 38558, 38560
- \l__coffin_aligned_internal_-
coffin 37603, 38261, 38268
- __coffin_attach:NnnNnnnnN
..... 38209, 38211, 38217, 38221
- __coffin_attach_mark:NnnNnnnn ..
.. 38209, 38242, 38423, 38439, 38455
- \l_coffin_bottom_corner_dim ...
..... 37889, 37921, 37925,
38004, 38015, 38016, 38036, 38044
- \l__coffin_bounding_prop
..... 37885, 37912, 37941,
37943, 37949, 37951, 37960, 38023
- \l__coffin_bounding_shift_dim ...
.. 37888, 37920, 38022, 38028, 38029
- __coffin_calculate_intersection:Nnn
.. 37781, 37781, 38253, 38256, 38540
- __coffin_calculate_intersection:nnnnnn
..... 37781, 37846, 37854
- __coffin_calculate_intersection:nnnnnnnn
..... 37781, 37786, 37797, 38491
- \c__coffin_corners_prop
..... 37355, 37430, 37637, 37644
- \l_coffin_corners_prop
.... 37886, 37903, 37907, 37930,

- 37935, 37966, 38006, 38033, 38080,
38084, 38090, 38096, 38131, 38145
- \l__coffin_cos_fp
1461, 1463, 37883, 37902, 37987, 37996
- __coffin_display_attach:Nnnnn ..
.. 38461, 38496, 38513, 38532, 38538
- \l__coffin_display_coffin
.... 38358, 38478, 38484, 38549,
38550, 38555, 38557, 38559, 38560
- \l__coffin_display_coord_coffin .
..... 38358, 38425,
38440, 38456, 38499, 38514, 38533
- \l__coffin_display_font_tl
..... 38403, 38428, 38502
- __coffin_display_handles_
aux:nnnn 38461, 38519, 38524, 38530
- __coffin_display_handles_
aux:nnnnnn ... 38461, 38482, 38486
- \l__coffin_display_handles_prop .
.. 38361, 38431, 38435, 38505, 38509
- \l__coffin_display_offset_dim ...
.. 38398, 38457, 38458, 38534, 38535
- \l__coffin_display_pole_coffin ..
.. 38358, 38418, 38424, 38463, 38497
- \l__coffin_display_poles_prop ...
..... 38402, 38468,
38473, 38477, 38479, 38481, 38488
- \l__coffin_display_x_dim
..... 38400, 38494, 38544
- \l__coffin_display_y_dim
..... 38400, 38495, 38546
- \c__coffin_empty_coffin 38348, 38353
- \l__coffin_error_bool
..... 37376, 37783, 37790,
37804, 37826, 37857, 38490, 38492
- __coffin_find_bounding_shift: ..
..... 37915, 38020, 38020
- __coffin_find_bounding_shift_
aux:nn 38020, 38024, 38026
- __coffin_find_corner_maxima:N ..
..... 37914, 38000, 38000
- __coffin_find_corner_maxima_
aux:nn 38000, 38007, 38009
- __coffin_greset_structure:N ...
..... 37419, 37634, 37641, 37702
- __coffin_gset_pole:Nnn
..... 37471, 37548, 37648, 37689
- __coffin_gupdate_corners:N
..... 37703, 37706, 37708
- __coffin_gupdate_poles:N
..... 37704, 37737, 37739
- __coffin_if_exist:NTF ... 37396,
37396, 37407, 37416, 37438, 37451,
37476, 37511, 37524, 37553, 37581,
37593, 37656, 37674, 38570, 38601
- __coffin_join:NnnNnnnnN
..... 38158, 38160, 38166, 38170
- \l__coffin_left_corner_dim
..... 37889, 37920, 37928,
38005, 38011, 38012, 38035, 38043
- __coffin_mark_handle_aux:nnnnNnn
..... 38416, 38444, 38449, 38453
- __coffin_offset_corner:Nnnnn ...
..... 38289, 38292, 38294
- __coffin_offset_corners:Nnn ...
..... 38195,
38196, 38202, 38203, 38289, 38289
- __coffin_offset_pole:Nnnnnnn ...
..... 38270, 38273, 38275
- __coffin_offset_poles:Nnn
..... 38193, 38194, 38199,
38200, 38236, 38237, 38270, 38270
- \l__coffin_offset_x_dim
.... 37377, 38176, 38177, 38180,
38191, 38193, 38195, 38201, 38204,
38238, 38257, 38265, 38543, 38551
- \l__coffin_offset_y_dim
37377, 38194, 38196, 38201, 38204,
38238, 38259, 38266, 38545, 38552
- __coffin_pole:nNn 37618, 37620, 37624
- \c__coffin_poles_prop
..... 37362, 37432, 37639, 37646
- \l__coffin_poles_prop
..... 37886, 37905, 37909,
37932, 37937, 37974, 38041, 38082,
38086, 38092, 38098, 38137, 38152
- __coffin_reset_structure:N 37410,
37634, 37634, 37696, 38184, 38228
- __coffin_resize:NnnNN
..... 38052, 38054, 38061, 38066
- __coffin_resize_common:NnnN ...
..... 38076, 38078, 38078, 38117
- \l__coffin_right_corner_dim
.. 37889, 37928, 38003, 38013, 38014
- __coffin_rotate:NnNNN
..... 37893, 37894, 37897, 37899
- __coffin_rotate_bounding:nnn ...
..... 37913, 37957, 37957
- __coffin_rotate_corner:Nnnnn ...
..... 37908, 37957, 37963
- __coffin_rotate_pole:Nnnnnnn ...
..... 37910, 37969, 37969
- __coffin_rotate_vector:nnNN ...
..... 37959,
37965, 37971, 37972, 37981, 37981
- __coffin_rule:nn
..... 38411, 38411, 38421, 38466

```

\__coffin_scale:NnnNN .....
..... 38100, 38101, 38104, 38106
\__coffin_scale_corner:Nnnn ....
..... 38085, 38128, 38128
\__coffin_scale_pole:Nnnnnn ....
..... 38087, 38128, 38134
\__coffin_scale_vector:nnNN ....
..... 38121, 38121, 38130, 38136
\l__coffin_scale_x_fp 38048, 38068,
38088, 38108, 38110, 38116, 38124
\l__coffin_scale_y_fp ... 38048,
38070, 38109, 38110, 38114, 38126
\l__coffin_scaled_total_height_-
dim ..... 38050, 38113, 38118
\l__coffin_scaled_width_dim ....
..... 38050, 38115, 38118
\__coffin_set_bounding:N .....
..... 37911, 37939, 37939
\__coffin_set_horizontal_-
pole:NnnN 37648, 37649, 37652, 37654
\__coffin_set_pole:Nnn .....
.... 37465, 37541, 37648, 37684,
38282, 38328, 38332, 38340, 38344
\__coffin_set_vertical:NnnNNN ...
..... 37462, 37464, 37470, 37474
\__coffin_set_vertical:NnNNNNNw .
..... 37537, 37539, 37546, 37551
\__coffin_set_vertical_aux: ....
..... 37462, 37481, 37499, 37557
\__coffin_set_vertical_pole:NnnN
..... 37648, 37667, 37670, 37672
\__coffin_shift_corner:Nnnn ....
..... 37931, 38031, 38031
\__coffin_shift_pole:Nnnnnn ....
..... 37933, 38031, 38039
\__coffin_show:Nnnnn .....
..... 38587, 38594, 38597, 38599
\__coffin_show_structure:NN ....
.. 38562, 38563, 38566, 38568, 38603
\l__coffin_sin_fp .....
1461, 1463, 37883, 37901, 37988, 37995
\l__coffin_slope_A_fp ..... 37374
\l__coffin_slope_B_fp ..... 37374
\l__coffin_tmp_box . 37352, 37485,
37491, 37496, 37562, 37568, 37573,
37917, 37924, 37926, 37927, 37929
\l__coffin_tmp_dim .....
.... 37352, 37948, 37950, 37954,
38111, 38114, 38179, 38181, 38182
\l__coffin_tmp_tl ..... 37352,
38280, 38281, 38283, 38432, 38433,
38436, 38437, 38445, 38450, 38506,
38507, 38510, 38511, 38520, 38525
\__coffin_to_value:N 37383, 37383,
37388, 37427, 37428, 37429, 37431,
37584, 37585, 37586, 37587, 37596,
37597, 37598, 37599, 37621, 37636,
37638, 37643, 37645, 37658, 37676,
37686, 37691, 37713, 37744, 37904,
37906, 37934, 37936, 38081, 38083,
38095, 38097, 38187, 38231, 38234,
38272, 38291, 38298, 38469, 38581
\l__coffin_top_corner_dim .....
.. 37889, 37925, 38002, 38017, 38018
\__coffin_update_B:nnnnnnnnN ...
..... 38307, 38318, 38336
\__coffin_update_corners:N .....
..... 37697, 37706, 37706
\__coffin_update_corners:NN ....
..... 37706, 37707, 37709, 37710
\__coffin_update_corners:NNN ...
..... 37706, 37712, 37716
\__coffin_update_poles:N .....
.. 37698, 37737, 37737, 38190, 38235
\__coffin_update_poles:NN .....
..... 37737, 37738, 37740, 37741
\__coffin_update_poles:NNN .....
..... 37737, 37743, 37747
\__coffin_update_T:nnnnnnnnN ...
..... 38307, 38311, 38324
\__coffin_update_vertical_-
poles:NNN 38206, 38239, 38307, 38307
\l__coffin_x_dim ... 37379, 37793,
37802, 37828, 37859, 37877, 37959,
37961, 37965, 37967, 37971, 37976,
38130, 38132, 38136, 38139, 38254,
38258, 38277, 38285, 38494, 38541
\l__coffin_x_prime_dim .....
..... 37379, 37973,
37977, 38254, 38258, 38541, 38544
\__coffin_x_shift_corner:Nnnn ...
..... 38091, 38143, 38143
\__coffin_x_shift_pole:Nnnnnn ...
..... 38093, 38143, 38150
\l__coffin_y_dim ..... 37379,
37794, 37806, 37824, 37873, 37959,
37961, 37965, 37967, 37971, 37976,
38130, 38132, 38136, 38139, 38255,
38260, 38278, 38285, 38495, 38542
\l__coffin_y_prime_dim .....
..... 37379, 37973,
37978, 38255, 38260, 38542, 38546
color commands:
color.sc ..... 335
\color_ensure_current: .....
..... 331, 1450, 37442,
37455, 37513, 37526, 38629, 38629

```

- \color_export:nnN .. 337, 39496, 39496
- \color_export:nnnN . 337, 39496, 39506
- \color_fill:n 335, 39348, 39348
- \color_fill:nn 335, 39348, 39358
- \l_color_fixed_model_tl 335, 38764,
38766, 39164, 39167, 39170, 39172,
39176, 39221, 39222, 39228, 39247,
39249, 39377, 39381, 39383, 39500
- \color_group_begin: .. 331, 36537,
36541, 36546, 36553, 36558, 36566,
36572, 36586, 36592, 36599, 36604,
36617, 36619, 36621, 36623, 36644,
36681, 36691, 36705, 36729, 36734,
36741, 36746, 36753, 36758, 36766,
36772, 36782, 36788, 38627, 38627
- \color_group_end: 331,
36537, 36541, 36546, 36553, 36558,
36578, 36599, 36604, 36617, 36619,
36627, 36681, 36729, 36734, 36741,
36746, 36753, 36758, 38627, 38628
- \color_if_exist:n 38647
- \color_if_exist:nTF .. 334, 38647,
38757, 38790, 38850, 39463, 40256
- \color_if_exist_p:n 334, 38647
- \color_log:n 334, 40247, 40249
- \color_math:nn 336, 39254, 39254
- \color_math:nn(n) 1497
- \color_math:nnn ... 336, 39254, 39259
- \l_color_math_active_tl
..... 336, 39251, 39309
- \color_model_new:nnn 337, 39616, 39616
- \color_profile_apply:nn
..... 338, 40218, 40218
- \color_select:n 335, 38420, 38427,
38465, 38501, 39193, 39193, 39199
- \color_select:nn
..... 335, 39193, 39200, 39206
- \color_set:nn 334, 39374, 39374
- \color_set:nnn 334,
39374, 39420, 39483, 39484, 39485,
39486, 39487, 39488, 39489, 39490
- \color_set_eq:nn ... 334, 39374, 39461
- \color_show:n 334, 40247, 40247
- \color_stroke:n ... 335, 39348, 39353
- \color_stroke:nn ... 335, 39348, 39363
- color internal commands:
 - \g_color_alternative_model_prop
..... 39603, 39715, 39813
 - \g_color_alternative_values_-
prop 39608,
39730, 39744, 39754, 39968, 40070
 - __color_backend_devicen_-
init:nnn 39983
 - __color_backend_iccbased_-
device:nnn ... 40235, 40240, 40245
 - __color_backend_iccbased_-
init:nnn 40215
 - __color_backend_reset: 38635, 39329
 - __color_backend_separation_-
init:nnnnn ... 39731, 39745, 39755
 - __color_backend_separation_-
init_CIELAB:nnn 39783
 - __color_check_model:N
..... 38760, 39165, 39165
 - __color_check_model:nn
..... 39165, 39169, 39179
 - \g_color_colorants_prop
..... 39586, 39716, 40031
 - __color_convert:nnN 38665, 38665,
38667, 38779, 38875, 38886, 39172
 - __color_convert:nnnN ... 38665,
38666, 38669, 38701, 39247, 39529
 - __color_convert_cmyk_cmyk:w ...
..... 38665, 38739
 - __color_convert_cmyk_gray:w ...
..... 38665, 38731
 - __color_convert_cmyk_rgb:w ...
..... 38665, 38733
 - __color_convert_devicen_-
cmyk:nnnnnnnnn 39795, 40091, 40094
 - __color_convert_devicen_-
cmyk:nnnnw ... 39795, 40088, 40116
 - __color_convert_devicen_cmyk_-
aux:nnnnw ... 39795, 40098, 40105
 - __color_convert_devicen_-
gray:nnn ... 39795, 40123, 40126
 - __color_convert_devicen_gray:nw
..... 39795, 40120, 40137
 - __color_convert_devicen_gray_-
aux:nw 39795, 40128, 40131
 - __color_convert_devicen_-
rgb:nnnnnnnn ... 39795, 40144, 40147
 - __color_convert_devicen_-
rgb:nnnw 39795, 40141, 40167
 - __color_convert_devicen_rgb_-
aux:nnnw 39795, 40151, 40157
 - __color_convert_gray_cmyk:w ...
..... 38665, 38706
 - __color_convert_gray_gray:w ...
..... 38665, 38702
 - __color_convert_gray_rgb:w ...
..... 38665, 38704
 - __color_convert_rgb_cmyk:nnn ...
... 1484, 38665, 38714, 38719, 39087
 - __color_convert_rgb_cmyk:nnnn ...
..... 38665, 38721, 38724

- _color_convert_rgb_cmyk:w 38665, 38712
- _color_convert_rgb_gray:w 38665, 38708
- _color_convert_rgb_rgb:w 38665, 38710
- \l_color_current_tl 1480, 38627, 38630, 38641, 38748, 38751, 38798, 39187, 39191, 39195, 39197, 39202, 39204, 39257, 39262, 39266, 39268, 39330, 39337, 39338, 39350, 39351, 39355, 39356, 39360, 39361, 39365, 39366, 39470, 39472, 39493, 39495
- _color_draw:nnn 39348, 39351, 39356, 39361, 39366, 39368
- _color_export:nN 39496, 39502, 39509, 39511
- _color_export:nnnN 39496, 39512, 39513
- _color_export:nnnNN 39524, 39524, 39544
- _color_export_comma-sep-cmyk:Nw 39554
- \c_color_export_comma-sep-cmyk_-tl 39534
- _color_export_comma-sep-rgb:Nw 39559
- \c_color_export_comma-sep-rgb_-tl 39534
- _color_export_format_backend:nnN 39522, 39522
- _color_export_format_comma-sep-cmyk:nnN 39539
- _color_export_format_comma-sep-rgb:nnN 39539
- _color_export_format_space-sep-cmyk:nnN 39539
- _color_export_format_space-sep-rgb:nnN 39539
- _color_export_HTML:n 39559, 39565, 39566, 39567, 39570
- _color_export_HTML:Nw 39559, 39561
- \c_color_export_HTML_tl 39534
- _color_export_space-sep-cmyk:Nw 39554
- \c_color_export_space-sep-cmyk_-tl 39534
- _color_export_space-sep-rgb:Nw 39559
- \c_color_export_space-sep-rgb_-tl 39534
- _color_extract:nnN 38659, 38659, 38664, 38804, 38843, 38844, 38852, 38867
- \c_color_fallback_cmyk_tl ... 39583
- \c_color_fallback_gray_tl ... 39583
- \c_color_fallback_rgb_tl ... 39583
- _color_finalize_current: 39184, 39184, 39196, 39203
- \c_color_icc_colorspace_-signatures_prop 40171, 40197
- \l_color_ignore_error_bool 38646, 38833, 39408
- _color_math:nn 39254, 39256, 39261, 39264
- _color_math_scan:w 1499, 39270, 39272, 39272, 39303, 39324, 39340
- _color_math_scan_auxi: 39272, 39284, 39288
- _color_math_scan_auxii: 39272, 39304, 39307
- _color_math_scan_auxiii:N 39316, 39322
- _color_math_scan_end: 39272, 39280, 39319, 39327
- _color_math_script_aux:N 39332, 39344, 39347
- _color_math_scripts:Nw 39295, 39332, 39332
- \g_color_math_seq 39253, 39266, 39330, 39337
- _color_model:N 38657, 38657, 38748, 39187, 39392, 39414, 39453, 39470, 39493
- _color_model_convert:nnn 39660, 39757
- _color_model_devicen:n 39795, 39795
- _color_model_devicen:nn 39795, 39800, 39808
- _color_model_devicen:nnn 39795, 39842, 39844
- _color_model_devicen:nnnn 39795, 39846, 39849
- _color_model_devicen_colorant:n 39795, 39986, 40029
- _color_model_devicen_convert:n 39795, 40061, 40066
- _color_model_devicen_convert:nnn 39795
- _color_model_devicen_convert:nnnn 39795, 39859, 40033, 40037
- _color_model_devicen_convert:nnnnn 40041, 40046, 40051, 40053
- _color_model_devicen_convert:w 39795
- _color_model_devicen_convert_-aux:n 39795, 40069, 40073

- _color_model_devicen_convert_-aux:w 40074, 40075
- _color_model_devicen_convert_-cmyk:n 39795
- _color_model_devicen_convert_-cmyk:nnn 40038
- _color_model_devicen_convert_-gray:n 39795
- _color_model_devicen_convert_-gray:nnn 40043
- _color_model_devicen_convert_-rgb:n 39795
- _color_model_devicen_convert_-rgb:nnn 40048
- _color_model_devicen_init:nnn 39795, 39858, 39947
- _color_model_devicen_init:nnnn 39795, 39949, 39960
- _color_model_devicen_mix:nw ... 39795, 39919, 39938, 39944
- _color_model_devicen_parse:nw ... 39795, 39914, 39924, 39933
- _color_model_devicen_parse_-1:nn 39795
- _color_model_devicen_parse_-2:nn 39795
- _color_model_devicen_parse_-3:nn 39795
- _color_model_devicen_parse_-4:nn 39795
- _color_model_devicen_parse_-generic:nn ... 39795, 39856, 39909
- _color_model_devicen_transform:nnn .. 39795, 40006, 40010, 40015, 40017
- _color_model_devicen_transform:w 39795, 39970, 39999
- _color_model_devicen_transform_-1:nnnnn 39795
- _color_model_devicen_transform_-3:nnnnn 39795
- _color_model_devicen_transform_-4:nnnnn 39795
- _color_model_iccbased:n 40182, 40182
- _color_model_iccbased:nn 40182, 40187, 40195
- _color_model_iccbased:nnn .. 40182
- _color_model_iccbased_aux:nnn 40182
- _color_model_iccbased_aux:nnnnn 40200, 40208
- _color_model_init:nnn .. 39639, 39639, 39659, 39709, 39851, 40210
- \g_color_model_int 39582, 39641, 39647, 40234, 40239, 40244
- _color_model_new:nnn 39616, 39618, 39622
- \c_color_model_range_CIELAB_tl . 39602
- _color_model_separation:n 39660, 39660
- _color_model_separation:nn ... 39660, 39665, 39673
- _color_model_separation:nnn ... 39660, 39678, 39686
- _color_model_separation:w 39660, 39693, 39706
- _color_model_separation_-CIELAB:nnnnnn 39660, 39767
- _color_model_separation_-CIELAB:nnnnnnn 39660, 39776, 39779
- _color_model_separation_-cmyk:nnnnnn 39660, 39719
- _color_model_separation_-gray:nnnnnn 39660, 39748
- _color_model_separation_-rgb:nnnnnn 39660, 39734
- \l_color_model_tl 38741, 38776, 38777, 38780, 38804, 38807, 38845, 38853, 38855, 38862, 38867, 38873, 38875, 38877, 38883, 38888, 39170, 39172, 39181, 39810, 39816, 39817, 39819, 39837, 39842
- \c_color_model_whitepoint_-CIELAB_a_tl 39595
- \c_color_model_whitepoint_-CIELAB_b_tl 39595
- \c_color_model_whitepoint_-CIELAB_d50_tl 39595
- \c_color_model_whitepoint_-CIELAB_d55_tl 39595
- \c_color_model_whitepoint_-CIELAB_d65_tl 39595
- \c_color_model_whitepoint_-CIELAB_d75_tl 39595
- \c_color_model_whitepoint_-CIELAB_e_tl 39595
- \l_color_named_.prop 39491
- \l_color_named_.tl 39491
- \l_color_named_tl . 39373, 39389, 39392, 39395, 39452, 39453, 39457
- \l_color_named_white_prop ... 39652
- \l_color_next_model_tl .. 38741, 38852, 38853, 38873, 38874, 38887
- \l_color_next_value_tl .. 38741, 38852, 38862, 38878, 38884, 38889


```

__color_parse:nN .....
    .... 38745, 38745, 39195, 39257,
    39350, 39355, 39389, 39410, 39501
__color_parse:Nw 38745, 38759, 38788
__color_parse_aux:nN .....
    .... 38745, 38752, 38755
__color_parse_break:w .....
    .... 38745, 38868, 38892
__color_parse_end: .....
    .... 38745, 38829, 38892, 38893
__color_parse_eq:Nn ..... 38745
__color_parse_eq:nNn ..... 38745
__color_parse_gray:n .....
    .... 38745, 38856, 38871
__color_parse_loop:nn .....
    .... 38745, 38821, 38848
__color_parse_loop:w .....
    .... 38745, 38805, 38811, 38828
__color_parse_loop_check:nn ...
    .... 38745, 38825, 38831
__color_parse_loop_init:Nnn ...
    .... 38745, 38794, 38801
__color_parse_mix:Nnnn .....
    .... 38745, 38861, 38894, 38900
__color_parse_mix:nNnn .....
    .... 38745, 38896, 38901
__color_parse_mix_cmyk:nw .....
    .... 38745, 38915, 39907
__color_parse_mix_gray:nw .....
    .... 38745, 38906, 39710, 39868
__color_parse_mix_rgb:nw .....
    .... 38745, 38908, 39892
__color_parse_model_&spot:w . 39117
__color_parse_model_cbrtlms:nnn
    .... 39134, 39136, 39141
__color_parse_model_cmy:w .....
    .... 39084, 39084
__color_parse_model_cmyk:w .....
    .... 38923, 38934
__color_parse_model_Gray:w .....
    .... 38948, 38948
__color_parse_model_gray:w .....
    .... 38923, 38923
__color_parse_model_hsb:nnn ...
    .... 38948,
    38951, 38954, 38957, 38994, 39091
__color_parse_model_hsb:nnnn . 38948
__color_parse_model_hsb:nnnnn 38948
__color_parse_model_HSB:w .....
    .... 38948, 38992
__color_parse_model_Hsb:w .....
    .... 38948, 38952
__color_parse_model_hsb:w .....
    .... 38948, 38950
__color_parse_model_hsb_0:nnnn .
    .... 38948
__color_parse_model_hsb_1:nnnn .
    .... 38948
__color_parse_model_hsb_2:nnnn .
    .... 38948
__color_parse_model_hsb_3:nnnn .
    .... 38948
__color_parse_model_hsb_4:nnnn .
    .... 38948
__color_parse_model_hsb_5:nnnn .
    .... 38948
__color_parse_model_hsb_aux:nnn
    .... 38948, 38961, 38965, 39077
__color_parse_model_hsb_-
    aux:nnnn ..... 38967, 38971
__color_parse_model_hsb_-
    aux:nnnnn ..... 38975, 38983
__color_parse_model_HTML:w ....
    .... 38948, 38999
__color_parse_model_HTML_aux:w .
    .... 39000, 39001
__color_parse_model_linearrgb:nnn
    .... 39119, 39119, 39150
__color_parse_model_linearrgb_-
    aux:n .....
    .. 39119, 39123, 39124, 39125, 39128
__color_parse_model_lms:nnn ...
    .... 39134, 39143, 39148
__color_parse_model_oklab:nnn ..
    .... 39134, 39134, 39156, 39159
__color_parse_model_oklab:w ...
    .... 39134, 39155
__color_parse_model_oklch:w ...
    .... 39157, 39157
__color_parse_model_RGB:w .....
    .... 38948, 39010
__color_parse_model_rgb:w .....
    .... 38923, 38925
__color_parse_model_tHsb:n ....
    .... 39089, 39092, 39094
__color_parse_model_tHsb:nw ...
    .... 39089, 39096, 39107, 39111
__color_parse_model_tHsb:w .....
    .... 39089, 39089
__color_parse_model_wave:w .....
    .... 38948, 39019
__color_parse_model_wave_-
    auxi:nn .....
    .. 38948, 39024, 39028, 39029, 39033
__color_parse_model_wave_-
    auxii:nn .. 38948, 39037, 39044,
    39051, 39058, 39065, 39069, 39075

```

- _color_parse_model_wave_rho:n 38948, 39038, 39045,
39052, 39059, 39066, 39080, 39082
- _color_parse_number:n 38923, 38924,
38929, 38930, 38931, 38938, 38939,
38940, 38941, 38944, 38976, 39712,
39867, 39873, 39887, 39888, 39889,
39901, 39902, 39903, 39904, 39931
- _color_parse_number:w 38923, 38945, 38946
- _color_parse_set_eq:Nn 38758, 38762, 38793
- _color_parse_set_eq:nNn 38765, 38766, 38769
- _color_parse_std:n 38745, 38857, 38880
- _color_profile_apply:nn 40218, 40220, 40223
- _color_profile_apply_cmyk:n 40218, 40242
- _color_profile_apply_gray:n 40218, 40232
- _color_profile_apply_rgb:n 40218, 40237
- _color_select:N 38630,
38632, 38632, 39197, 39204, 39338
- _color_select:nn 38632, 38634, 38638, 38639
- _color_select:nnN 39193, 39219, 39229, 39236, 39452
- _color_select_loop:Nw 39193, 39223, 39225, 39233
- _color_select_main:Nnn 39193, 39202,
39207, 39262, 39360, 39365, 39508
- _color_select_main:Nw 39193, 39211, 39216
- _color_select_math:N 38632, 38637, 39268
- _color_select_swap:Nnn 39193, 39232, 39245
- _color_set:nn 39374, 39382, 39385
- _color_set:nnn 39374, 39376, 39379
- _color_set:nnw 39374, 39396, 39399
- _color_set_aux:nnn 39374, 39426, 39430
- _color_set_colon:nnw 39374, 39432, 39437
- _color_set_loop:nw 39374, 39443, 39444, 39447, 39458
- _color_show:n 40247, 40258, 40267
- _color_show:Nn 40247, 40248, 40250, 40251
- _color_tmp:w 39540,
39548, 39549, 39550, 39551, 39552
- \l_color_tmp_int 38643, 39964, 39967, 40023
- \l_color_tmp_prop 39581, 39631, 39662,
39675, 39690, 39769, 39797, 40184
- \l_color_tmp_tl 38643, 38806, 38809,
39403, 39410, 39412, 39414, 39415,
39453, 39455, 39456, 39663, 39666,
39676, 39679, 39691, 39693, 39770,
39774, 39777, 39798, 39801, 39814,
39817, 39819, 39962, 39973, 39996,
40019, 40185, 40188, 40198, 40201
- \l_color_value_tl 38741, 38773, 38774, 38778, 38780,
38784, 38804, 38807, 38845, 38859,
38862, 38867, 38876, 39173, 39176,
39182, 39247, 39249, 39969, 39971
- _color_values:N 38657, 38658, 38751, 39191,
39395, 39415, 39457, 39472, 39495
- \columnwidth 37506
- \compoundhyphenmode 797
- \contextversion 10271, 10301, 10524, 10544
- \copy 179
- \copyfont 933
- cos 283
- cosd 284
- cot 283
- cotd 284
- \count 180, 20331
- \countdef 181
- \cr 182
- \crampeddisplaystyle 799
- \crampedscriptscriptstyle 800
- \crampedscriptstyle 802
- \crampedtextstyle 803
- \crrcr 183
- \creationdate 768
- cs commands:
 - \cs:w 23, 570, 594, 682,
905, 907, 959, 1401, 1403, 1423,
1425, 1486, 1864, 1908, 1939, 2134,
2211, 2454, 2496, 2505, 2507, 2511,
2512, 2513, 2561, 2567, 2573, 2579,
2613, 2615, 2620, 2627, 2628, 2682,
2686, 2726, 3047, 4321, 7181, 7184,
8657, 8659, 11058, 11197, 12217,
14558, 14564, 18303, 18415, 19148,
19151, 20078, 21094, 21141, 21293,
21589, 21910, 22060, 22152, 22790,
22791, 23489, 24394, 24413, 24481,

25312, 25502, 25534, 25952, 25978,
 25991, 26025, 26067, 26631, 26647,
 28387, 29492, 30591, 30609, 32090,
 32328, 32627, 32636, 35733, 36395
 \cs_argument_spec:N . . . 41431, 41432
 \cs_end: 23, 443, 594, 682, 683, 905,
 907, 959, 1401, 1404, 1423, 1425,
 1429, 1486, 1840, 1853, 1864, 1882,
 1897, 1908, 1922, 1933, 1939, 2062,
 2134, 2211, 2454, 2496, 2505, 2507,
 2511, 2512, 2513, 2561, 2567, 2573,
 2579, 2613, 2615, 2620, 2627, 2628,
 2682, 2686, 2726, 3047, 4322, 4327,
 6990, 7197, 8654, 8660, 8662, 8664,
 8666, 8668, 8670, 8672, 8674, 8676,
 8678, 8680, 11058, 11073, 11076,
 11077, 11186, 11197, 12217, 14564,
 14567, 18303, 18415, 19103, 19128,
 19139, 19148, 19151, 20078, 21092,
 21094, 21139, 21141, 21293, 21589,
 21910, 22060, 22152, 22502, 22790,
 22791, 23489, 24397, 24413, 24489,
 25315, 25506, 25538, 25958, 25984,
 25997, 26028, 26070, 26637, 26653,
 28387, 29495, 30591, 30615, 32095,
 32330, 32632, 32641, 35733, 36395
 \cs_generate_from_arg_count:NNnn
 21, 2114, 2114, 2124,
 2125, 2126, 2127, 2157, 3173, 3173
 \cs_generate_variant:Nn
 16, 34–36, 67, 436, 437,
 2760, 2760, 2773, 2774, 3099, 3173,
 3174, 3175, 3176, 3287, 3289, 3311,
 3314, 3320, 3326, 4090, 5104, 6255,
 7005, 7046, 7375, 7376, 7399, 7401,
 8369, 8375, 8384, 8385, 8386, 8387,
 8390, 8391, 8402, 8403, 8406, 8409,
 8429, 8440, 8442, 8591, 8592, 8597,
 8598, 9028, 9060, 9183, 9208, 9335,
 9338, 9547, 9549, 9551, 9553, 9821,
 10241, 10242, 10243, 10244, 10282,
 10287, 10321, 10346, 10364, 10366,
 10368, 10539, 10560, 10572, 10580,
 10596, 10608, 10610, 10612, 10639,
 10642, 10643, 10656, 10662, 10663,
 10666, 10671, 10777, 11135, 11178,
 11288, 11302, 11305, 11318, 11328,
 11372, 11390, 11393, 11396, 11399,
 11429, 11497, 11504, 11517, 11530,
 11560, 11577, 11583, 11630, 12251,
 12257, 12258, 12263, 12264, 12269,
 12270, 12275, 12276, 12293, 12294,
 12311, 12312, 12313, 12314, 12315,
 12316, 12317, 12318, 12381, 12382,
 12383, 12384, 12385, 12386, 12387,
 12388, 12389, 12390, 12391, 12392,
 12449, 12450, 12451, 12452, 12453,
 12454, 12455, 12456, 12457, 12458,
 12459, 12460, 12475, 12509, 12510,
 12511, 12512, 12650, 12659, 12661,
 12663, 12665, 12667, 12669, 12671,
 12673, 12733, 12736, 12739, 12742,
 12755, 12766, 12767, 12772, 12773,
 12774, 12775, 12935, 12938, 12968,
 12978, 12999, 13004, 13006, 13015,
 13027, 13028, 13065, 13066, 13067,
 13074, 13075, 13076, 13089, 13090,
 13091, 13092, 13093, 13094, 13158,
 13169, 13369, 13380, 13381, 13404,
 13411, 13415, 13492, 13513, 13515,
 13534, 13550, 13604, 13610, 13633,
 13636, 13697, 13712, 13713, 13716,
 13717, 13747, 13748, 13749, 13750,
 13751, 13752, 13753, 13762, 13763,
 13764, 13765, 13798, 13799, 13804,
 13805, 13884, 13919, 13948, 13966,
 13992, 14006, 14054, 14115, 14193,
 14212, 14221, 14232, 14252, 14267,
 14284, 14285, 14286, 14299, 14395,
 14415, 14441, 14448, 16628, 16629,
 16630, 16637, 16760, 16820, 16827,
 17098, 17099, 17169, 17176, 17200,
 17388, 17391, 17394, 17397, 17400,
 17429, 17430, 17431, 17432, 17433,
 17434, 17440, 17480, 17481, 17482,
 17483, 17484, 17485, 17498, 17501,
 17504, 17507, 17510, 17513, 17526,
 17539, 17540, 17562, 17563, 17564,
 17565, 17570, 17571, 17572, 17573,
 17590, 17591, 17616, 17617, 17618,
 17619, 17625, 17626, 17702, 17703,
 17751, 17752, 17802, 17815, 17816,
 17834, 17860, 17861, 17913, 17919,
 17947, 17976, 17986, 18009, 18010,
 18067, 18111, 18134, 18148, 18150,
 18151, 18153, 18154, 18168, 18170,
 18305, 18308, 18329, 18344, 18345,
 18350, 18351, 18353, 18355, 18368,
 18369, 18370, 18371, 18380, 18381,
 18382, 18383, 18388, 18389, 18392,
 18401, 18404, 18413, 18703, 18724,
 19061, 19065, 19092, 19100, 19112,
 19114, 19116, 19146, 19149, 19152,
 19237, 19238, 19278, 19279, 19280,
 19281, 19295, 19296, 19305, 19306,
 19307, 19308, 19318, 19319, 19320,
 19321, 19331, 19333, 19335, 19337,
 19349, 19374, 19375, 19402, 19404,

19427, 19428, 19466, 19467, 19472,
 19473, 19564, 19572, 19602, 19605,
 19637, 19666, 19687, 19711, 19720,
 19722, 19779, 19786, 19795, 19828,
 19830, 19832, 19984, 19985, 19986,
 19987, 20746, 20809, 20831, 20834,
 20837, 20866, 20869, 20872, 20875,
 20878, 20881, 20938, 20956, 20970,
 20973, 20993, 20996, 21016, 21022,
 21028, 21034, 21068, 21069, 21070,
 21121, 21187, 21188, 21201, 21202,
 21203, 21204, 21229, 21234, 21236,
 21241, 21243, 21248, 21250, 21255,
 21257, 21264, 21399, 21414, 21437,
 21451, 21471, 21473, 21591, 21597,
 21601, 21602, 21607, 21608, 21617,
 21618, 21621, 21624, 21632, 21633,
 21641, 21642, 22025, 22044, 22052,
 22062, 22068, 22071, 22072, 22077,
 22078, 22087, 22088, 22090, 22092,
 22097, 22098, 22103, 22104, 22133,
 22134, 22136, 22154, 22160, 22165,
 22166, 22171, 22172, 22181, 22182,
 22184, 22186, 22191, 22192, 22197,
 22198, 22204, 22352, 22353, 22529,
 22631, 22634, 22650, 22706, 22718,
 22814, 22935, 22941, 23192, 23203,
 23206, 23209, 23220, 23223, 23229,
 23240, 23243, 23249, 23293, 23409,
 23411, 23634, 23655, 23689, 23803,
 23855, 23888, 23899, 23937, 23940,
 23950, 24032, 24034, 24064, 24106,
 24115, 24124, 24133, 24196, 24198,
 24745, 24748, 24755, 26456, 26463,
 26464, 26465, 26468, 26469, 26472,
 26473, 26478, 26479, 26486, 26487,
 26488, 26489, 26491, 26493, 26797,
 26855, 30021, 30075, 30153, 30198,
 30213, 30267, 31138, 31158, 31187,
 31241, 31249, 31257, 31319, 31320,
 31407, 31408, 31409, 31410, 31419,
 31420, 31439, 31440, 31442, 31458,
 31460, 31462, 31476, 31479, 31558,
 31597, 31603, 31616, 31675, 31693,
 31749, 31805, 32078, 32662, 33004,
 33331, 33365, 33368, 33371, 33374,
 33636, 33833, 34943, 35294, 35515,
 35669, 35717, 36397, 36402, 36403,
 36408, 36409, 36414, 36415, 36420,
 36421, 36429, 36430, 36431, 36434,
 36440, 36443, 36449, 36452, 36458,
 36461, 36464, 36465, 36493, 36494,
 36502, 36505, 36508, 36517, 36535,
 36548, 36549, 36560, 36561, 36574,
 36575, 36594, 36595, 36614, 36615,
 36736, 36737, 36748, 36749, 36760,
 36761, 36774, 36775, 36790, 36791,
 36794, 36795, 36798, 36804, 36819,
 36822, 36940, 36946, 36986, 36989,
 37006, 37009, 37026, 37029, 37043,
 37046, 37064, 37067, 37093, 37096,
 37102, 37108, 37173, 37179, 37208,
 37215, 37222, 37239, 37242, 37245,
 37248, 37251, 37299, 37302, 37413,
 37422, 37435, 37448, 37461, 37467,
 37473, 37521, 37534, 37543, 37550,
 37590, 37602, 37650, 37653, 37668,
 37671, 37895, 37898, 38058, 38065,
 38102, 38105, 38163, 38169, 38214,
 38220, 38357, 38460, 38537, 38564,
 38567, 38589, 38592, 38595, 38598,
 38664, 38667, 38668, 38701, 38900,
 39199, 39206, 39659, 40037, 40474,
 40490, 40499, 40509, 40610, 40631,
 40904, 40984, 41060, 41061, 41067,
 41436, 41456, 41459, 41462, 41484,
 41589, 41672, 41674, 41789, 42330
 \cs_gset:Nn 20, 2129, 2206
 .cs_gset:Np 248, 22980
 \cs_gset:Npe 18, 1448,
 1453, 1455, 1996, 2015, 2019, 17956
 \cs_gset:Npn . . . 15, 18, 1448, 1451,
 1486, 1493, 1495, 1496, 1497, 1498,
 1499, 1500, 1501, 1502, 1503, 1504,
 1505, 1506, 1507, 1508, 1509, 1510,
 1511, 1512, 1513, 1514, 1515, 1516,
 1517, 1518, 1519, 1520, 1521, 1522,
 1523, 1524, 1525, 1526, 1527, 1528,
 1529, 1530, 1531, 1532, 1533, 1534,
 1535, 1536, 1537, 1538, 1539, 1540,
 1541, 1542, 1543, 1544, 1545, 1546,
 1547, 1548, 1549, 1550, 1552, 1553,
 1554, 1555, 1556, 1557, 1558, 1559,
 1560, 1577, 1578, 1579, 1580, 1589,
 1591, 1593, 1598, 1619, 1670, 1673,
 1735, 1736, 1737, 1738, 1788, 1790,
 1792, 1794, 1799, 1805, 1806, 1810,
 1817, 1820, 1847, 1863, 1891, 1907,
 1910, 1912, 1914, 1916, 1920, 1927,
 1931, 1938, 1941, 1948, 1950, 1952,
 1971, 1995, 2015, 2018, 3637, 7022,
 9330, 9332, 9381, 11861, 17250,
 17251, 17289, 17951, 22989, 22991,
 25192, 31073, 31078, 32104, 33838
 \cs_gset:Npx
 18, 1448, 1455, 1997, 2015, 2020
 \cs_gset_eq:NN 22, 1742,
2042, 2046, 2047, 2048, 2049, 2059,

- 8381, 8383, 9235, 10361, 10605,
12228, 12230, 12249, 14717, 14721,
17386, 17728, 17961, 17966, 19982,
20807, 20836, 20880, 20972, 20985,
20989, 21019, 21060, 21184, 21198,
21214, 21402, 21410, 30896, 30976,
31307, 31313, 32421, 32431, 42024
\cs_gset_nopar:Nn 20, 2129, 2206
\cs_gset_nopar:Npe
. . . 18, 714, 754, 759, 1448, 1449,
1993, 2004, 2010, 12245, 12255,
13587, 13607, 13635, 13725, 20818,
20825, 20836, 20880, 20972, 20995,
21019, 21026, 21032, 21184, 21198,
21214, 21224, 21228, 23791, 42135
\cs_gset_nopar:Npn
. 18, 1448, 1448, 1485, 1581, 1582,
1992, 2004, 2009, 17058, 17357, 41870
\cs_gset_nopar:Npx
. . . . 18, 1448, 1450, 1994, 2004, 2011
\cs_gset_protected:Nn 21, 2129, 2206
.cs_gset_protected:Np . . . 248, 22980
\cs_gset_protected:Npe
. 18, 1448, 1463, 1465, 2002,
2033, 2037, 18688, 21875, 26867, 41389
\cs_gset_protected:Npn
. 18, 1448, 1461, 1469, 1471, 1474,
1476, 1479, 1481, 1487, 1562, 1563,
1569, 1575, 1576, 1583, 1595, 1597,
1599, 1601, 1603, 1604, 1605, 1607,
1616, 1618, 1620, 1622, 1624, 1625,
1626, 1628, 1637, 1649, 1675, 1692,
1711, 1719, 1727, 1739, 1741, 1743,
1745, 1757, 1771, 1808, 1954, 1967,
1969, 1973, 1981, 1986, 2001, 2033,
2036, 3238, 3246, 3259, 3679, 3701,
3981, 7627, 8951, 10304, 10475,
10547, 11870, 12570, 12572, 12574,
12607, 12611, 12956, 13720, 13952,
18020, 18677, 19586, 20986, 21405,
21868, 22993, 22995, 26860, 31781,
35485, 35495, 41188, 41383, 41400,
41409, 41649, 41655, 41661, 41687,
41695, 41726, 41731, 42134, 42135
\cs_gset_protected:Npx
. . . . 18, 1448, 1465, 2003, 2033, 2038
\cs_gset_protected_nopar:Nn
. 21, 2129, 2206
\cs_gset_protected_nopar:Npe . . .
19, 1448, 1458, 1460, 1999, 2024, 2028
\cs_gset_protected_nopar:Npn . . .
. . . . 19, 1448, 1456, 1998, 2024, 2027
\cs_gset_protected_nopar:Npx . . .
. . . . 19, 1448, 1460, 2000, 2024, 2029
\cs_if_eq:NN 2250, 12820, 20222
\cs_if_eq:NNTF 29,
1553, 2250, 2256, 2257, 2258, 2260,
2261, 2262, 2264, 2265, 2266, 5730,
9281, 9391, 20884, 20976, 22836,
24954, 24965, 24993, 24995, 24997,
25197, 30851, 30992, 33125, 33175,
33195, 33639, 35635, 41383, 41409
\cs_if_eq_p:NN
. 29, 2250, 2255, 2259, 2263, 5747,
32679, 33264, 33265, 35698, 35699
\cs_if_exist:N . . 1824, 1838, 1851,
8462, 8464, 12295, 12296, 17541,
17543, 18356, 18358, 19122, 19124,
19297, 19299, 21330, 21332, 21609,
21611, 22079, 22081, 22173, 22175,
24191, 24193, 26587, 26588, 31297,
31299, 31750, 31752, 36422, 36424
\cs_if_exist:NNTF 23,
29, 382, 642, 787, 943, 1566, 1572,
1824, 1836, 1878, 1911, 1913, 1915,
1917, 1918, 2270, 2338, 2358, 2378,
3098, 3236, 3254, 3257, 4376, 4382,
4388, 5108, 5458, 7167, 8736, 8737,
8738, 8740, 8744, 8776, 8820, 8841,
8945, 9079, 9107, 9279, 9316, 10271,
10275, 10301, 10524, 10528, 10544,
10672, 11026, 11207, 11648, 11701,
11822, 12568, 14322, 14323, 14332,
14688, 14697, 14701, 14715, 18331,
18332, 18333, 18334, 19073, 19074,
20295, 20314, 22543, 22657, 22760,
22766, 22788, 23332, 23371, 23379,
23398, 23414, 23420, 23437, 23447,
23462, 23467, 23561, 23629, 23645,
23775, 24074, 25190, 25364, 26446,
30853, 30889, 30909, 30969, 30994,
31008, 31035, 31625, 31726, 31779,
31819, 32324, 32579, 33243, 33627,
33692, 33700, 34906, 34913, 35688,
35788, 37386, 37388, 38676, 39238,
39624, 39629, 39688, 40635, 40816,
41133, 41799, 41808, 42015, 42382
\cs_if_exist_p:N
. 29, 381, 1824, 3634, 8753,
31702, 33211, 33273, 33411, 35647,
37502, 38405, 41139, 41140, 41141
\cs_if_exist_use:N 23, 408,
1910, 1916, 1952, 6119, 9701, 9719,
10746, 23416, 23441, 33439, 33479
\cs_if_exist_use:NNTF
. 23, 1910, 1910, 1912, 1914,
1920, 1931, 1948, 1949, 1950, 1951,
1953, 3009, 3078, 4670, 4677, 5067,

- 5072, 5116, 5526, 5612, 7103, 9250,
 17203, 23385, 24620, 25322, 25324,
 33477, 33743, 33749, 33807, 33809,
 35105, 35357, 35466, 38673, 38687,
 39515, 39855, 40225, 41691, 41699
 \cs_if_free:N 1866, 1880, 1895
 \cs_if_free:NTF 29, 65, 642,
 1866, 1975, 2935, 2962, 9691, 42368
 \cs_if_free_p:N 28, 29, 65, 1866
 \cs_log:N 22, 419, 2295, 2298, 2299, 2300
 \cs_meaning:N
 . 22, 393, 1410, 1411, 1426, 1427,
 1434, 1437, 2307, 9076, 31791, 42062
 \cs_new:Nn 19, 66, 2129, 2206
 \cs_new:Npe 16, 42, 435,
 1984, 1996, 2015, 2022, 2786, 3639,
 3884, 4167, 4691, 4693, 4695, 4697,
 4699, 4701, 8424, 8430, 9132, 9671,
 9673, 9675, 9682, 10719, 10731,
 11258, 14328, 15765, 16657, 16672,
 21692, 21697, 21702, 22608, 23690,
 24378, 25223, 25808, 27028, 29087,
 29093, 29732, 30568, 30707, 32828,
 32879, 33205, 39759, 39916, 40056
 \cs_new:Npn . . . 15, 16, 21, 66, 441,
 456, 1553, 1600, 1621, 1984, 1995,
 2015, 2021, 2100, 2102, 2104, 2112,
 2165, 2255, 2256, 2257, 2258, 2259,
 2260, 2261, 2262, 2263, 2264, 2265,
 2266, 2333, 2340, 2348, 2360, 2368,
 2380, 2388, 2424, 2427, 2436, 2437,
 2447, 2448, 2449, 2450, 2451, 2452,
 2453, 2455, 2457, 2459, 2472, 2478,
 2484, 2495, 2497, 2504, 2506, 2508,
 2515, 2516, 2518, 2520, 2522, 2524,
 2529, 2534, 2540, 2546, 2552, 2558,
 2564, 2570, 2576, 2582, 2589, 2596,
 2603, 2610, 2617, 2624, 2633, 2634,
 2636, 2641, 2646, 2648, 2658, 2659,
 2661, 2663, 2665, 2667, 2669, 2675,
 2681, 2683, 2689, 2691, 2698, 2705,
 2706, 2707, 2708, 2709, 2710, 2712,
 2721, 2723, 2726, 2727, 2728, 2730,
 2732, 2737, 2747, 2750, 2755, 2756,
 2757, 2758, 2827, 2848, 2870, 2873,
 2881, 2894, 2909, 2920, 2952, 3043,
 3045, 3301, 3457, 3470, 3475, 3481,
 3482, 3489, 3496, 3503, 3510, 3517,
 3518, 3520, 3527, 3533, 3627, 3632,
 3648, 3654, 3858, 3863, 3870, 3906,
 3912, 3917, 3933, 3956, 3963, 4020,
 4026, 4044, 4049, 4064, 4066, 4068,
 4075, 4091, 4093, 4096, 4102, 4361,
 4394, 4399, 4401, 4408, 4410, 4411,
 4416, 4422, 4444, 4446, 4448, 4668,
 4674, 4685, 4690, 4703, 4708, 4720,
 4735, 4745, 4757, 4780, 4785, 4886,
 4901, 4907, 4924, 4934, 5446, 5728,
 5743, 5777, 5793, 5840, 5878, 5909,
 5915, 5921, 5929, 5934, 5940, 5945,
 5959, 5974, 5983, 5991, 5993, 6045,
 6054, 6125, 6161, 6368, 6782, 6886,
 6889, 6910, 6912, 6918, 6920, 6927,
 6937, 7136, 7526, 7642, 7648, 7687,
 7694, 8327, 8412, 8474, 8475, 8484,
 8486, 8496, 8501, 8506, 8508, 8516,
 8517, 8518, 8519, 8520, 8521, 8522,
 8523, 8524, 8525, 8535, 8551, 8561,
 8577, 8587, 8589, 8593, 8595, 8599,
 8607, 8612, 8620, 8627, 8629, 8631,
 8633, 8635, 8640, 8646, 8649, 8656,
 8658, 8660, 8661, 8663, 8665, 8667,
 8669, 8671, 8673, 8675, 8677, 8679,
 8681, 8686, 8687, 8688, 8689, 8690,
 8691, 8692, 8693, 8694, 8695, 8707,
 8710, 9114, 9168, 9311, 9380, 9415,
 9477, 9482, 9487, 9489, 9491, 9493,
 9499, 9507, 9513, 9519, 9652, 9654,
 9689, 9822, 9844, 9846, 9848, 10210,
 10211, 10220, 10233, 10235, 10237,
 10239, 10459, 10461, 10536, 10674,
 10682, 10720, 10737, 10792, 10844,
 10853, 10872, 10873, 10881, 10887,
 10895, 10905, 10910, 10916, 10922,
 10995, 10997, 10999, 11047, 11055,
 11061, 11068, 11069, 11075, 11077,
 11084, 11089, 11097, 11107, 11109,
 11118, 11120, 11121, 11123, 11173,
 11179, 11184, 11192, 11201, 11216,
 11222, 11226, 11232, 11234, 11245,
 11246, 11247, 11249, 11267, 11269,
 11300, 11303, 11306, 11311, 11316,
 11319, 11321, 11329, 11343, 11353,
 11363, 11370, 11375, 11382, 11452,
 11558, 11561, 11572, 11578, 11584,
 11589, 11595, 11609, 11647, 11715,
 11837, 11838, 11842, 11843, 12504,
 12565, 12728, 12796, 12881, 12884,
 12885, 12886, 12887, 12899, 12928,
 12936, 12939, 12946, 12952, 12969,
 12976, 12979, 12987, 13000, 13002,
 13005, 13007, 13016, 13021, 13026,
 13029, 13040, 13041, 13042, 13043,
 13050, 13057, 13059, 13061, 13063,
 13068, 13070, 13072, 13097, 13106,
 13115, 13122, 13125, 13133, 13135,
 13140, 13145, 13148, 13153, 13159,
 13160, 13161, 13162, 13170, 13212,

13221, 13240, 13242, 13255, 13262,
13278, 13289, 13297, 13303, 13306,
13311, 13323, 13329, 13330, 13332,
13340, 13346, 13353, 13355, 13357,
13370, 13372, 13374, 13382, 13390,
13396, 13403, 13405, 13410, 13414,
13416, 13417, 13425, 13437, 13446,
13455, 13460, 13466, 13489, 13490,
13491, 13493, 13547, 13670, 13678,
13685, 13686, 13862, 13867, 13872,
13877, 13882, 13891, 13897, 13902,
13907, 13912, 13917, 13921, 13927,
13929, 13937, 13939, 13941, 13967,
13993, 13995, 13997, 14005, 14007,
14018, 14027, 14030, 14041, 14050,
14053, 14055, 14063, 14065, 14072,
14093, 14103, 14108, 14113, 14114,
14116, 14124, 14126, 14134, 14140,
14146, 14165, 14167, 14176, 14182,
14189, 14191, 14194, 14204, 14211,
14213, 14222, 14227, 14233, 14244,
14251, 14253, 14259, 14261, 14266,
14268, 14274, 14275, 14280, 14281,
14282, 14283, 14287, 14292, 14297,
14300, 14302, 14310, 14315, 14325,
14343, 14354, 14356, 14362, 14371,
14386, 14396, 14400, 14414, 14416,
14496, 14504, 14511, 14552, 14554,
14560, 14566, 14568, 14573, 14578,
14594, 14610, 14624, 14723, 14729,
14755, 14761, 14793, 14803, 14814,
14840, 14847, 14907, 14914, 14936,
14946, 15015, 15025, 15062, 15124,
15165, 15167, 15184, 15190, 15213,
15243, 15264, 15273, 15353, 15373,
15393, 15416, 15423, 15450, 15553,
15567, 15594, 15603, 15605, 15626,
15631, 15637, 15642, 15710, 15733,
15745, 15752, 15758, 15763, 16636,
16644, 16652, 16666, 16680, 16686,
16692, 16698, 16704, 16710, 16719,
16726, 16749, 16759, 16761, 16770,
16803, 16811, 16821, 16828, 16835,
16836, 16838, 16847, 16883, 16891,
16893, 16902, 16940, 16949, 16951,
16957, 16970, 16985, 16991, 17003,
17008, 17017, 17034, 17039, 17040,
17041, 17068, 17074, 17082, 17089,
17096, 17100, 17106, 17139, 17149,
17152, 17256, 17265, 17312, 17317,
17319, 17328, 17334, 17339, 17367,
17374, 17472, 17478, 17561, 17574,
17672, 17679, 17697, 17770, 17800,
17828, 17833, 17855, 17890, 17892,
17900, 17906, 17914, 17920, 17922,
17924, 17935, 17977, 17987, 18012,
18027, 18037, 18045, 18047, 18054,
18060, 18088, 18106, 18110, 18112,
18135, 18136, 18137, 18144, 18146,
18186, 18206, 18211, 18213, 18215,
18221, 18229, 18235, 18237, 18245,
18253, 18261, 18269, 18282, 18284,
18291, 18293, 18415, 18422, 18436,
18441, 18447, 18458, 18463, 18470,
18472, 18474, 18476, 18478, 18480,
18482, 18492, 18494, 18496, 18498,
18500, 18502, 18504, 18506, 18508,
18510, 18512, 18514, 18524, 18529,
18534, 18539, 18544, 18546, 18552,
18570, 18578, 18586, 18592, 18598,
18606, 18614, 18620, 18626, 18633,
18652, 18662, 18664, 18704, 18718,
18725, 18757, 18789, 18791, 18793,
18799, 18805, 18817, 18825, 18837,
18845, 18878, 18911, 18913, 18915,
18917, 18919, 18924, 18929, 18934,
18939, 18940, 18941, 18942, 18943,
18944, 18945, 18946, 18947, 18948,
18949, 18950, 18951, 18952, 18953,
18954, 18955, 18964, 18965, 18974,
18980, 18982, 18991, 18998, 19004,
19006, 19008, 19024, 19035, 19058,
19136, 19137, 19145, 19147, 19150,
19156, 19157, 19158, 19159, 19160,
19161, 19162, 19163, 19164, 19165,
19166, 19184, 19185, 19186, 19188,
19194, 19200, 19230, 19231, 19271,
19373, 19463, 19465, 19474, 19481,
19484, 19497, 19503, 19541, 19551,
19558, 19565, 19573, 19580, 19613,
19623, 19631, 19638, 19644, 19654,
19656, 19658, 19667, 19670, 19679,
19688, 19712, 19713, 19716, 19718,
19723, 19733, 19744, 19746, 19749,
19755, 19756, 19764, 19780, 19787,
19796, 19798, 19812, 19814, 19815,
19816, 19818, 19823, 19870, 19940,
19946, 19952, 19958, 19990, 19996,
20026, 20068, 20093, 20095, 20243,
20273, 20350, 20363, 20448, 20460,
20461, 20469, 20478, 20487, 20496,
20498, 20500, 20502, 20504, 20506,
20508, 20510, 20512, 20514, 20516,
20518, 20520, 20527, 20533, 20540,
20541, 20542, 20543, 20546, 20640,
20648, 20650, 20652, 20662, 20672,
20757, 20760, 20762, 20763, 20769,
20783, 20794, 20801, 21050, 21089,

21102, 21104, 21116, 21172, 21174,
21343, 21370, 21377, 21387, 21415,
21425, 21438, 21440, 21442, 21450,
21452, 21468, 21497, 21503, 21585,
21644, 21649, 21651, 21659, 21667,
21675, 21677, 21684, 21686, 21688,
21713, 21719, 21732, 21734, 21736,
21738, 21740, 21748, 21753, 21758,
21763, 21768, 21770, 21776, 21778,
21786, 21794, 21800, 21806, 21814,
21822, 21828, 21834, 21841, 21855,
21889, 21891, 21897, 21910, 21911,
21918, 21926, 21931, 21945, 21954,
21959, 21969, 21979, 21997, 22003,
22009, 22018, 22023, 22120, 22123,
22128, 22131, 22199, 22228, 22239,
22245, 22247, 22249, 22259, 22265,
22270, 22277, 22285, 22289, 22296,
22298, 22306, 22310, 22317, 22327,
22335, 22343, 22357, 22366, 22378,
22379, 22380, 22381, 22383, 22399,
22410, 22418, 22423, 22429, 22447,
22448, 22559, 22576, 22580, 22617,
22918, 22924, 23162, 23168, 23483,
23485, 23550, 23559, 23565, 23567,
23572, 23576, 23580, 23584, 23590,
23602, 23606, 23610, 23627, 23643,
23701, 23713, 23926, 23928, 23938,
23961, 24033, 24035, 24037, 24048,
24107, 24109, 24116, 24122, 24140,
24148, 24157, 24167, 24217, 24218,
24219, 24220, 24221, 24222, 24223,
24225, 24227, 24228, 24238, 24262,
24264, 24266, 24275, 24277, 24284,
24296, 24297, 24299, 24309, 24319,
24329, 24339, 24347, 24349, 24356,
24358, 24359, 24364, 24371, 24385,
24387, 24403, 24404, 24412, 24414,
24423, 24425, 24437, 24442, 24447,
24452, 24454, 24456, 24458, 24460,
24467, 24469, 24477, 24479, 24491,
24493, 24495, 24497, 24521, 24523,
24525, 24526, 24527, 24529, 24531,
24533, 24536, 24554, 24569, 24570,
24576, 24592, 24598, 24732, 24733,
24734, 24735, 24736, 24737, 24738,
24743, 24746, 24749, 24803, 24805,
24807, 24809, 24815, 24818, 24820,
24829, 24830, 24839, 24852, 24865,
24872, 24886, 24902, 24914, 24925,
24935, 24941, 24952, 24962, 24991,
25002, 25019, 25030, 25035, 25055,
25057, 25068, 25073, 25086, 25109,
25110, 25114, 25131, 25132, 25156,
25164, 25182, 25211, 25237, 25241,
25244, 25246, 25252, 25264, 25276,
25283, 25289, 25297, 25320, 25335,
25354, 25362, 25377, 25392, 25403,
25413, 25423, 25428, 25437, 25454,
25467, 25473, 25479, 25481, 25488,
25518, 25546, 25562, 25573, 25578,
25596, 25614, 25625, 25640, 25645,
25656, 25666, 25676, 25692, 25736,
25741, 25748, 25756, 25762, 25767,
25771, 25788, 25796, 25828, 25845,
25859, 25878, 25886, 25895, 25904,
25917, 25919, 25935, 25945, 25946,
25963, 25970, 25975, 25988, 26001,
26006, 26034, 26048, 26076, 26077,
26081, 26098, 26120, 26122, 26133,
26165, 26169, 26184, 26201, 26225,
26227, 26229, 26231, 26241, 26246,
26257, 26269, 26280, 26293, 26313,
26331, 26333, 26345, 26351, 26359,
26373, 26380, 26391, 26398, 26412,
26500, 26509, 26513, 26538, 26549,
26558, 26583, 26585, 26602, 26624,
26629, 26640, 26657, 26684, 26685,
26686, 26687, 26703, 26714, 26722,
26734, 26740, 26746, 26754, 26762,
26768, 26774, 26782, 26790, 26798,
26812, 26839, 26887, 26893, 26904,
26928, 26931, 26933, 26935, 26943,
26947, 26954, 26961, 26962, 26963,
26964, 26965, 26967, 26970, 26972,
27001, 27009, 27020, 27022, 27024,
27026, 27033, 27057, 27059, 27069,
27084, 27093, 27107, 27115, 27123,
27130, 27137, 27145, 27155, 27169,
27182, 27184, 27190, 27207, 27214,
27216, 27223, 27228, 27245, 27246,
27248, 27267, 27273, 27283, 27295,
27302, 27317, 27325, 27363, 27373,
27394, 27396, 27398, 27407, 27419,
27432, 27447, 27461, 27474, 27482,
27500, 27518, 27526, 27534, 27545,
27546, 27555, 27556, 27565, 27575,
27589, 27599, 27610, 27618, 27620,
27631, 27637, 27672, 27693, 27695,
27697, 27699, 27706, 27715, 27720,
27727, 27734, 27754, 27759, 27776,
27787, 27792, 27802, 27804, 27814,
27822, 27824, 27830, 27832, 27837,
27844, 27863, 27864, 27869, 27877,
27879, 27902, 27915, 27922, 27930,
27931, 27932, 27933, 27934, 27935,
27943, 27950, 27952, 27954, 27976,
27981, 27991, 28002, 28013, 28026,

28037, 28042, 28049, 28059, 28061,
28070, 28079, 28093, 28095, 28097,
28112, 28122, 28127, 28136, 28144,
28152, 28158, 28167, 28169, 28182,
28187, 28195, 28200, 28210, 28216,
28223, 28230, 28237, 28239, 28245,
28247, 28252, 28254, 28268, 28278,
28290, 28295, 28302, 28312, 28314,
28316, 28336, 28350, 28364, 28384,
28397, 28399, 28404, 28417, 28422,
28431, 28437, 28447, 28460, 28492,
28493, 28495, 28497, 28499, 28504,
28518, 28525, 28534, 28553, 28559,
28569, 28588, 28596, 28629, 28635,
28644, 28646, 28660, 28719, 28727,
28745, 28763, 28765, 28770, 28795,
28818, 28846, 28862, 28873, 28884,
28905, 28920, 28925, 28930, 28932,
28946, 28952, 28968, 28976, 28986,
28996, 29009, 29027, 29033, 29047,
29062, 29100, 29102, 29104, 29106,
29108, 29123, 29138, 29153, 29168,
29183, 29198, 29206, 29220, 29222,
29228, 29240, 29248, 29255, 29481,
29488, 29525, 29533, 29534, 29545,
29552, 29554, 29560, 29571, 29581,
29588, 29595, 29610, 29666, 29679,
29720, 29726, 29733, 29753, 29755,
29772, 29787, 29800, 29807, 29812,
29814, 29823, 29836, 29839, 29861,
29874, 29889, 29907, 29922, 29932,
29941, 29954, 29970, 29987, 30000,
30006, 30008, 30015, 30016, 30018,
30019, 30022, 30027, 30033, 30038,
30040, 30063, 30071, 30073, 30076,
30081, 30087, 30092, 30094, 30117,
30142, 30151, 30152, 30154, 30159,
30161, 30166, 30168, 30178, 30186,
30194, 30196, 30199, 30204, 30209,
30211, 30212, 30214, 30219, 30224,
30226, 30231, 30238, 30252, 30257,
30259, 30269, 30271, 30273, 30275,
30277, 30288, 30298, 30300, 30303,
30308, 30310, 30318, 30319, 30333,
30340, 30346, 30347, 30360, 30375,
30381, 30403, 30418, 30428, 30449,
30458, 30481, 30499, 30510, 30515,
30528, 30546, 30551, 30579, 30583,
30588, 30595, 30601, 30606, 30618,
30637, 30644, 30646, 30662, 30668,
30675, 30683, 30695, 30715, 30725,
30731, 30736, 30749, 30761, 30769,
30778, 30801, 30811, 31006, 31089,
31104, 31105, 31153, 31159, 31173,
31242, 31250, 31258, 31264, 31271,
31276, 31282, 31294, 31421, 31430,
31435, 31443, 31607, 31713, 31715,
31722, 31734, 31738, 31742, 31917,
31923, 31934, 31946, 31960, 31979,
31996, 32002, 32007, 32051, 32053,
32055, 32058, 32061, 32067, 32073,
32079, 32082, 32088, 32098, 32100,
32111, 32116, 32117, 32464, 32484,
32572, 32577, 32586, 32587, 32590,
32593, 32595, 32598, 32606, 32615,
32619, 32625, 32634, 32646, 32650,
32652, 32668, 32674, 32676, 32687,
32702, 32712, 32729, 32743, 32760,
32773, 32775, 32776, 32825, 32843,
32854, 32856, 32858, 32869, 32903,
32918, 32919, 32921, 32923, 32987,
32995, 33002, 33005, 33007, 33013,
33024, 33036, 33041, 33050, 33064,
33075, 33085, 33090, 33096, 33123,
33137, 33142, 33147, 33158, 33160,
33165, 33171, 33185, 33191, 33217,
33227, 33237, 33239, 33250, 33256,
33261, 33269, 33270, 33285, 33286,
33299, 33304, 33314, 33321, 33355,
33357, 33359, 33361, 33363, 33366,
33369, 33372, 33375, 33377, 33386,
33395, 33397, 33406, 33419, 33430,
33436, 33437, 33444, 33455, 33457,
33465, 33473, 33484, 33486, 33492,
33498, 33504, 33509, 33515, 33529,
33540, 33549, 33556, 33563, 33573,
33579, 33588, 33593, 33598, 33609,
33611, 33625, 33634, 33637, 33644,
33649, 33654, 33659, 33670, 33677,
33682, 33684, 33688, 33690, 33710,
33717, 33725, 33741, 33747, 33753,
33760, 33770, 33783, 33796, 33803,
33805, 33814, 33823, 33828, 33845,
33849, 33870, 33874, 33885, 33891,
33939, 33944, 33945, 33947, 33960,
33996, 34001, 34011, 34027, 34038,
34051, 34069, 34074, 34105, 34112,
34126, 34137, 34149, 34159, 34175,
34181, 34215, 34223, 34233, 34246,
34275, 34309, 34387, 34405, 34422,
34447, 34463, 34473, 34483, 34495,
34507, 34520, 34526, 34536, 34549,
34557, 34566, 34573, 34582, 34584,
34597, 34610, 34614, 34624, 34634,
34647, 34672, 34682, 34689, 34694,
34716, 34729, 34739, 34746, 34751,
34763, 34770, 34781, 34788, 34803,
34817, 34828, 34833, 34853, 34937,

34944, 34955, 34962, 34968, 34978,
 34984, 34998, 35009, 35023, 35027,
 35031, 35044, 35056, 35069, 35079,
 35095, 35103, 35109, 35116, 35142,
 35144, 35146, 35148, 35155, 35159,
 35164, 35170, 35193, 35198, 35203,
 35209, 35220, 35231, 35237, 35244,
 35251, 35261, 35279, 35288, 35295,
 35301, 35303, 35304, 35309, 35315,
 35321, 35326, 35331, 35336, 35345,
 35378, 35380, 35389, 35406, 35414,
 35422, 35427, 35435, 35440, 35446,
 35455, 35506, 35516, 35523, 35525,
 35527, 35533, 35544, 35545, 35550,
 35555, 35562, 35573, 35578, 35580,
 35582, 35587, 35592, 35603, 35614,
 35619, 35626, 35631, 35642, 35644,
 35668, 35670, 35679, 35685, 35695,
 35707, 35733, 35786, 36080, 36086,
 36092, 36098, 36102, 36111, 36132,
 36134, 36143, 36145, 36166, 36175,
 36180, 36200, 36205, 36224, 36226,
 36238, 36240, 36259, 36261, 36263,
 36275, 36280, 36289, 36316, 36348,
 36363, 36369, 36375, 36377, 36379,
 36390, 37618, 37624, 38657, 38658,
 38702, 38704, 38706, 38708, 38710,
 38712, 38719, 38724, 38731, 38733,
 38739, 38894, 38901, 38906, 38908,
 38915, 38923, 38925, 38934, 38944,
 38946, 38948, 38950, 38952, 38957,
 38965, 38971, 38983, 38985, 38986,
 38987, 38988, 38989, 38990, 38991,
 38992, 38999, 39001, 39010, 39019,
 39033, 39075, 39082, 39084, 39089,
 39094, 39107, 39117, 39119, 39128,
 39134, 39141, 39148, 39155, 39157,
 39513, 39570, 39711, 39722, 39729,
 39737, 39743, 39751, 39753, 39785,
 39787, 39866, 39872, 39874, 39883,
 39896, 39911, 39924, 39938, 40029,
 40055, 40066, 40073, 40075, 40088,
 40094, 40105, 40120, 40126, 40131,
 40141, 40147, 40157, 40212, 40213,
 40267, 40700, 40905, 40910, 40956,
 40985, 40990, 41031, 41039, 41041,
 41068, 41069, 41073, 41081, 41093,
 41120, 41122, 41123, 41314, 41323,
 41336, 41347, 41432, 41487, 41489,
 41491, 41493, 41513, 41515, 41517,
 41519, 41521, 41523, 41525, 41527,
 41535, 41537, 41548, 41550, 41552,
 41554, 41557, 41560, 41564, 41567,
 41570, 41573, 41576, 41579, 41582,
 41584, 41586, 41588, 41600, 41602,
 41604, 41606, 41608, 41610, 41612,
 41614, 41616, 41618, 41620, 41622,
 41624, 41626, 41628, 41630, 41633,
 41636, 41638, 41681, 41684, 41790,
 41898, 41899, 41929, 41934, 41939,
 41948, 41961, 41965, 41976, 41999
 \cs_new:Npx . 16, 1984, 1997, 2015, 2023
 \cs_new_eq:NN 21, 67, 410, 412, 938,
 1744, 2042, 2050, 2055, 2056, 2057,
 2426, 2435, 2465, 2725, 2753, 2754,
 2993, 3180, 3605, 3606, 4357, 4358,
 4364, 4377, 4383, 4389, 4512, 4513,
 4514, 4613, 4621, 4642, 4681, 4682,
 4683, 4684, 4883, 6654, 6919, 7316,
 7848, 8366, 8368, 8388, 8389, 8722,
 8723, 8724, 8725, 8728, 8729, 8730,
 8731, 9070, 10281, 10303, 10393,
 10538, 10546, 10675, 10811, 11172,
 11441, 11484, 11545, 11551, 11832,
 11833, 11834, 12244, 12245, 12729,
 12730, 13412, 13413, 13696, 13710,
 13711, 13714, 13715, 13793, 13816,
 13887, 13888, 13889, 13890, 14052,
 14430, 14435, 14442, 14772, 14788,
 14790, 15779, 17066, 17353, 17356,
 17381, 17401, 17402, 17403, 17404,
 17405, 17406, 17407, 17408, 17620,
 18011, 18149, 18152, 18155, 18156,
 18157, 18158, 18159, 18160, 18199,
 18200, 18201, 18202, 18203, 18214,
 18335, 18336, 18339, 18414, 18666,
 18667, 18668, 18702, 19060, 19064,
 19089, 19180, 19233, 19234, 19239,
 19240, 19241, 19242, 19243, 19244,
 19245, 19246, 19247, 19248, 19249,
 19250, 19251, 19252, 19253, 19254,
 19401, 19403, 19721, 19988, 19989,
 20147, 20149, 20150, 20151, 20153,
 20156, 20157, 20536, 20537, 20538,
 20739, 21580, 21581, 21582, 21643,
 21909, 22024, 22028, 22029, 22069,
 22070, 22111, 22125, 22126, 22127,
 22130, 22135, 22139, 22140, 22201,
 22202, 22203, 22207, 22208, 22238,
 23408, 23773, 24007, 24008, 24213,
 24214, 24215, 24216, 24422, 24591,
 24885, 24913, 24921, 24922, 24923,
 24932, 24934, 25735, 25854, 25855,
 25856, 26455, 26466, 26467, 30266,
 30268, 30686, 30689, 30860, 31001,
 31375, 31441, 32684, 33463, 33686,
 33715, 33836, 33887, 33889, 33958,
 33994, 34471, 34612, 34863, 34865,

- 35154, 35157, 35158, 35243, 35250,
 35314, 35320, 36389, 36426, 36427,
 36428, 36462, 36463, 36474, 36475,
 36476, 36581, 36612, 36613, 36776,
 36777, 36792, 36793, 37383, 37610,
 37611, 37612, 37613, 37614, 37615,
 37616, 37617, 38627, 38628, 39710,
 39868, 39892, 39907, 41199, 41740
 \cs_new_nopar:Nn 19, 2129, 2206
 \cs_new_nopar:Npe
 16, 1984, 1993, 2004, 2013
 \cs_new_nopar:Npn
 . 16, 410, 411, 1984, 1992, 2004, 2012
 \cs_new_nopar:Npx
 16, 1984, 1994, 2004, 2014
 \cs_new_protected:Nn . 19, 2129, 2206
 \cs_new_protected:Npe
 . . 16, 435, 440, 1984, 2002, 2033,
 2040, 2775, 2779, 2784, 2943, 2947,
 4191, 5286, 5300, 5302, 9539, 9541,
 9543, 9545, 10322, 11148, 11518,
 12746, 17517, 19091, 19978, 20687,
 23314, 37499, 38745, 39184, 39430,
 39542, 39644, 39650, 40538, 40764
 \cs_new_protected:Npn
 16, 441, 1553, 1606, 1627,
 1984, 2001, 2033, 2039, 2042, 2043,
 2044, 2045, 2046, 2047, 2048, 2049,
 2050, 2055, 2056, 2057, 2058, 2060,
 2069, 2090, 2114, 2124, 2126, 2137,
 2146, 2268, 2277, 2279, 2281, 2283,
 2285, 2293, 2295, 2296, 2298, 2299,
 2301, 2309, 2311, 2313, 2438, 2466,
 2631, 2653, 2720, 2744, 2760, 2773,
 2791, 2795, 2798, 2807, 2931, 2948,
 2958, 2967, 2991, 2997, 3005, 3016,
 3018, 3020, 3022, 3024, 3035, 3037,
 3050, 3058, 3069, 3088, 3090, 3092,
 3094, 3096, 3106, 3196, 3215, 3222,
 3234, 3267, 3286, 3288, 3290, 3309,
 3312, 3315, 3321, 3327, 3343, 3352,
 3372, 3382, 3393, 3403, 3413, 3414,
 3421, 3427, 3437, 3447, 3543, 3549,
 3551, 3566, 3659, 3670, 3690, 3712,
 3722, 3724, 3748, 3755, 3767, 3770,
 3773, 3783, 3791, 3798, 3807, 3822,
 3839, 3850, 3970, 3972, 3979, 3988,
 3994, 3996, 4001, 4011, 4013, 4015,
 4103, 4122, 4133, 4152, 4175, 4187,
 4212, 4223, 4237, 4245, 4252, 4254,
 4264, 4274, 4305, 4311, 4313, 4318,
 4334, 4359, 4362, 4365, 4367, 4379,
 4385, 4391, 4450, 4452, 4453, 4458,
 4464, 4472, 4482, 4496, 4515, 4533,
 4535, 4543, 4555, 4574, 4576, 4581,
 4589, 4591, 4598, 4603, 4605, 4607,
 4614, 4622, 4624, 4626, 4628, 4635,
 4640, 4643, 4649, 4895, 4953, 4965,
 4976, 4989, 5022, 5051, 5060, 5065,
 5070, 5075, 5096, 5105, 5112, 5121,
 5126, 5133, 5146, 5148, 5150, 5152,
 5158, 5180, 5193, 5220, 5225, 5243,
 5257, 5292, 5313, 5315, 5323, 5325,
 5327, 5329, 5331, 5333, 5337, 5348,
 5364, 5377, 5383, 5394, 5407, 5413,
 5432, 5452, 5483, 5494, 5509, 5522,
 5540, 5548, 5553, 5555, 5557, 5574,
 5593, 5595, 5618, 5630, 5650, 5671,
 5678, 5685, 5696, 5703, 5709, 5767,
 5819, 5828, 5841, 5856, 5865, 5884,
 5903, 6060, 6131, 6141, 6143, 6145,
 6152, 6197, 6210, 6226, 6228, 6230,
 6235, 6250, 6256, 6279, 6299, 6314,
 6321, 6328, 6330, 6332, 6339, 6353,
 6369, 6378, 6392, 6404, 6421, 6430,
 6432, 6444, 6453, 6465, 6478, 6485,
 6505, 6536, 6570, 6588, 6597, 6603,
 6609, 6615, 6658, 6667, 6681, 6700,
 6726, 6731, 6741, 6753, 6791, 6800,
 6812, 6819, 6821, 6823, 6843, 6848,
 6854, 6865, 6870, 6875, 6891, 6943,
 6961, 6963, 7006, 7030, 7047, 7053,
 7055, 7075, 7101, 7112, 7121, 7130,
 7163, 7177, 7188, 7194, 7203, 7211,
 7246, 7252, 7255, 7263, 7269, 7272,
 7281, 7284, 7287, 7290, 7295, 7304,
 7307, 7310, 7315, 7321, 7326, 7331,
 7336, 7337, 7338, 7346, 7347, 7348,
 7371, 7373, 7377, 7385, 7387, 7389,
 7393, 7394, 7413, 7430, 7432, 7434,
 7436, 7453, 7455, 7457, 7472, 7480,
 7490, 7501, 7510, 7527, 7539, 7549,
 7558, 7598, 7635, 7637, 7666, 7671,
 7702, 7724, 7742, 7744, 7771, 7773,
 7802, 7819, 7830, 7835, 7849, 7855,
 7857, 7858, 7864, 7866, 7867, 7873,
 7875, 7877, 7883, 7885, 7887, 7895,
 7915, 7921, 7927, 7933, 7941, 7943,
 7945, 7947, 7949, 7951, 7953, 7955,
 7957, 7962, 7990, 8000, 8005, 8014,
 8016, 8030, 8343, 8345, 8347, 8354,
 8368, 8370, 8376, 8378, 8380, 8382,
 8392, 8397, 8404, 8407, 8435, 8437,
 8439, 8441, 8443, 8718, 8863, 8880,
 8933, 8939, 8957, 8968, 8991, 9021,
 9025, 9053, 9057, 9061, 9066, 9118,
 9159, 9161, 9163, 9184, 9214, 9217,
 9219, 9226, 9236, 9242, 9319, 9327,

9336, 9339, 9346, 9387, 9416, 9436,
9441, 9452, 9528, 9530, 9563, 9586,
9642, 9656, 9699, 9721, 9722, 9735,
9740, 9766, 9775, 9777, 9779, 9796,
9823, 9825, 9827, 9828, 9830, 9832,
9834, 9836, 9838, 9840, 9842, 10281,
10285, 10299, 10310, 10331, 10337,
10353, 10365, 10367, 10369, 10381,
10382, 10383, 10410, 10412, 10423,
10425, 10444, 10446, 10448, 10463,
10465, 10467, 10473, 10480, 10489,
10491, 10493, 10498, 10538, 10542,
10554, 10561, 10573, 10581, 10587,
10597, 10609, 10611, 10613, 10625,
10626, 10627, 10637, 10640, 10644,
10650, 10657, 10664, 10665, 10667,
10668, 10669, 10670, 10683, 10714,
10725, 10743, 10778, 10801, 10814,
10833, 10837, 10926, 10942, 10951,
10959, 10968, 10975, 10981, 11130,
11166, 11281, 11283, 11388, 11391,
11394, 11397, 11416, 11424, 11492,
11498, 11505, 11506, 11511, 11531,
11546, 11552, 11619, 11624, 11631,
11632, 11633, 11660, 11673, 11685,
11695, 11697, 11699, 11708, 11716,
11839, 11840, 11845, 12223, 12225,
12227, 12229, 12231, 12236, 12246,
12252, 12259, 12261, 12265, 12267,
12271, 12273, 12277, 12285, 12303,
12305, 12307, 12309, 12319, 12324,
12329, 12334, 12342, 12350, 12355,
12360, 12365, 12373, 12393, 12395,
12400, 12405, 12413, 12421, 12423,
12428, 12433, 12441, 12469, 12476,
12478, 12480, 12482, 12494, 12513,
12528, 12546, 12591, 12593, 12622,
12640, 12651, 12653, 12655, 12657,
12675, 12697, 12703, 12731, 12734,
12737, 12740, 12762, 12764, 12768,
12770, 12854, 12855, 12856, 12953,
12966, 12993, 12995, 12997, 13077,
13079, 13081, 13083, 13085, 13087,
13376, 13378, 13512, 13514, 13516,
13532, 13535, 13548, 13551, 13584,
13586, 13588, 13590, 13599, 13605,
13611, 13628, 13629, 13631, 13634,
13637, 13648, 13653, 13658, 13666,
13668, 13718, 13722, 13727, 13732,
13737, 13742, 13754, 13756, 13758,
13760, 13766, 13781, 13794, 13796,
13800, 13802, 13949, 13964, 13973,
13983, 13985, 14436, 14443, 14452,
14584, 14600, 14616, 14622, 14626,
14628, 14648, 14668, 14676, 14686,
14695, 14779, 14787, 14789, 14791,
14801, 14807, 14831, 14842, 14848,
14851, 14853, 14858, 14884, 14890,
14896, 14924, 14998, 15046, 15099,
15182, 15188, 15211, 15241, 15262,
15338, 15434, 15439, 15441, 15443,
15519, 15521, 15523, 15528, 15538,
15617, 15622, 15624, 15677, 15679,
15681, 15686, 15696, 17055, 17157,
17159, 17170, 17177, 17183, 17201,
17210, 17215, 17220, 17225, 17230,
17236, 17241, 17248, 17254, 17263,
17273, 17275, 17277, 17282, 17287,
17299, 17344, 17383, 17389, 17392,
17395, 17398, 17409, 17414, 17419,
17424, 17435, 17441, 17443, 17445,
17447, 17449, 17486, 17488, 17490,
17496, 17499, 17502, 17505, 17508,
17511, 17535, 17537, 17545, 17553,
17566, 17568, 17576, 17578, 17580,
17592, 17594, 17596, 17621, 17623,
17633, 17639, 17652, 17661, 17686,
17688, 17690, 17715, 17716, 17717,
17742, 17774, 17782, 17792, 17803,
17805, 17807, 17809, 17817, 17835,
17837, 17839, 17948, 17953, 17958,
17964, 17970, 17999, 18017, 18068,
18070, 18072, 18078, 18080, 18082,
18167, 18169, 18171, 18300, 18306,
18309, 18342, 18343, 18346, 18348,
18352, 18354, 18360, 18362, 18364,
18366, 18372, 18374, 18376, 18378,
18384, 18386, 18390, 18393, 18402,
18405, 18416, 18669, 18671, 18673,
18680, 18682, 18684, 18696, 19062,
19066, 19090, 19095, 19101, 19110,
19113, 19115, 19117, 19153, 19154,
19155, 19167, 19168, 19169, 19187,
19235, 19255, 19257, 19259, 19282,
19284, 19286, 19301, 19303, 19309,
19311, 19313, 19322, 19324, 19326,
19339, 19347, 19350, 19352, 19354,
19362, 19390, 19407, 19409, 19411,
19429, 19431, 19433, 19468, 19470,
19514, 19581, 19597, 19603, 19606,
19608, 19829, 19831, 19833, 19851,
19852, 19853, 19868, 19872, 19874,
19876, 19878, 19880, 19882, 19884,
19886, 19888, 19890, 19892, 19894,
19896, 19898, 19900, 19902, 19904,
19906, 19908, 19910, 19912, 19914,
19916, 19918, 19920, 19922, 19924,
19926, 19928, 19930, 19932, 19934,

19936, 19938, 19942, 19944, 19948,
19950, 19954, 19956, 19960, 19972,
20547, 20549, 20551, 20556, 20563,
20572, 20583, 20588, 20602, 20610,
20628, 20630, 20632, 20634, 20636,
20638, 20698, 20715, 20720, 20727,
20732, 20734, 20748, 20749, 20755,
20758, 20777, 20804, 20810, 20815,
20832, 20835, 20838, 20844, 20851,
20856, 20864, 20867, 20870, 20873,
20876, 20879, 20882, 20895, 20901,
20911, 20920, 20926, 20939, 20944,
20957, 20968, 20971, 20974, 20983,
20991, 20994, 20997, 21003, 21009,
21011, 21017, 21023, 21029, 21035,
21040, 21051, 21056, 21057, 21063,
21082, 21122, 21136, 21148, 21156,
21175, 21181, 21189, 21195, 21221,
21223, 21225, 21227, 21271, 21289,
21296, 21304, 21320, 21400, 21470,
21472, 21474, 21510, 21517, 21539,
21546, 21554, 21564, 21586, 21592,
21598, 21599, 21603, 21605, 21613,
21615, 21619, 21622, 21625, 21627,
21634, 21636, 21742, 21864, 21871,
21883, 22026, 22030, 22038, 22045,
22053, 22057, 22063, 22073, 22075,
22083, 22085, 22089, 22091, 22093,
22095, 22099, 22101, 22137, 22141,
22149, 22155, 22161, 22163, 22167,
22169, 22177, 22179, 22183, 22185,
22187, 22189, 22193, 22195, 22205,
22209, 22437, 22493, 22500, 22508,
22530, 22535, 22540, 22553, 22563,
22588, 22594, 22629, 22632, 22635,
22651, 22653, 22655, 22671, 22682,
22684, 22686, 22704, 22707, 22709,
22719, 22733, 22746, 22753, 22771,
22773, 22775, 22794, 22796, 22801,
22815, 22824, 22851, 22860, 22869,
22902, 22925, 22936, 22942, 22944,
22946, 22948, 22950, 22952, 22954,
22956, 22958, 22960, 22962, 22964,
22966, 22968, 22970, 22972, 22974,
22976, 22978, 22980, 22982, 22984,
22986, 22988, 22990, 22992, 22994,
22996, 22998, 23000, 23002, 23004,
23006, 23008, 23010, 23012, 23014,
23016, 23018, 23020, 23022, 23024,
23026, 23028, 23030, 23032, 23034,
23036, 23038, 23040, 23042, 23044,
23046, 23048, 23050, 23052, 23054,
23056, 23058, 23060, 23062, 23064,
23066, 23068, 23070, 23072, 23074,
23076, 23078, 23080, 23082, 23084,
23086, 23088, 23090, 23092, 23094,
23096, 23098, 23100, 23102, 23104,
23106, 23108, 23110, 23112, 23114,
23116, 23118, 23120, 23122, 23124,
23126, 23128, 23130, 23132, 23174,
23176, 23182, 23193, 23204, 23207,
23210, 23221, 23224, 23230, 23241,
23244, 23250, 23258, 23263, 23268,
23294, 23299, 23303, 23309, 23323,
23330, 23344, 23367, 23393, 23410,
23412, 23426, 23435, 23457, 23493,
23517, 23549, 23656, 23658, 23660,
23667, 23787, 23793, 23878, 23880,
23941, 23976, 24002, 24011, 24020,
24055, 24057, 24065, 24076, 24084,
24091, 24097, 24125, 24134, 24176,
24181, 24195, 24197, 24199, 24229,
24232, 24343, 24618, 24635, 24637,
24639, 24641, 24671, 24673, 24675,
24677, 24701, 24703, 24705, 24707,
24709, 24711, 24713, 24715, 24717,
26454, 26457, 26459, 26461, 26470,
26471, 26474, 26476, 26480, 26481,
26482, 26483, 26484, 26490, 26492,
26494, 26568, 26570, 26856, 26863,
26875, 29765, 30629, 30838, 30849,
30865, 30869, 30875, 30880, 30884,
30900, 30904, 30962, 30964, 30979,
30984, 31025, 31030, 31110, 31112,
31126, 31139, 31178, 31188, 31200,
31205, 31215, 31223, 31229, 31304,
31310, 31321, 31330, 31332, 31334,
31336, 31338, 31376, 31377, 31379,
31381, 31383, 31385, 31411, 31415,
31459, 31461, 31463, 31474, 31477,
31480, 31519, 31524, 31531, 31537,
31539, 31561, 31563, 31575, 31586,
31598, 31614, 31619, 31632, 31645,
31653, 31662, 31676, 31687, 31694,
31775, 31788, 31795, 33326, 33893,
33898, 33900, 33902, 33904, 33909,
33914, 33916, 33918, 33920, 33925,
33931, 33933, 33935, 33937, 35482,
35492, 35712, 36392, 36398, 36400,
36404, 36406, 36410, 36412, 36416,
36418, 36432, 36435, 36441, 36444,
36450, 36453, 36459, 36466, 36468,
36470, 36472, 36489, 36491, 36500,
36503, 36506, 36509, 36511, 36518,
36536, 36538, 36543, 36550, 36555,
36562, 36568, 36576, 36582, 36588,
36596, 36601, 36606, 36608, 36610,
36616, 36618, 36620, 36622, 36624,

- 36632, 36634, 36636, 36638, 36646,
 36648, 36650, 36652, 36654, 36659,
 36674, 36676, 36678, 36683, 36685,
 36687, 36693, 36695, 36697, 36699,
 36707, 36709, 36711, 36713, 36718,
 36726, 36731, 36738, 36743, 36750,
 36755, 36762, 36768, 36778, 36784,
 36796, 36799, 36817, 36820, 36823,
 36833, 36867, 36878, 36889, 36900,
 36911, 36922, 36935, 36941, 36947,
 36962, 36969, 36979, 36984, 36987,
 36990, 37004, 37007, 37010, 37024,
 37027, 37030, 37041, 37044, 37047,
 37062, 37065, 37068, 37077, 37091,
 37094, 37097, 37103, 37109, 37122,
 37159, 37166, 37168, 37174, 37180,
 37209, 37216, 37223, 37240, 37243,
 37246, 37249, 37252, 37297, 37300,
 37303, 37396, 37405, 37414, 37423,
 37436, 37449, 37462, 37468, 37474,
 37509, 37522, 37535, 37536, 37537,
 37544, 37551, 37577, 37578, 37579,
 37591, 37634, 37641, 37648, 37651,
 37654, 37666, 37669, 37672, 37684,
 37689, 37694, 37700, 37706, 37708,
 37710, 37716, 37737, 37739, 37741,
 37747, 37781, 37797, 37854, 37893,
 37896, 37899, 37939, 37957, 37963,
 37969, 37981, 38000, 38009, 38020,
 38026, 38031, 38039, 38052, 38059,
 38066, 38078, 38100, 38103, 38106,
 38121, 38128, 38134, 38143, 38150,
 38158, 38164, 38170, 38209, 38215,
 38221, 38242, 38251, 38270, 38275,
 38289, 38294, 38307, 38324, 38336,
 38350, 38411, 38416, 38453, 38461,
 38486, 38530, 38538, 38562, 38565,
 38568, 38587, 38590, 38593, 38596,
 38599, 38629, 38632, 38637, 38639,
 38659, 38665, 38669, 38755, 38762,
 38769, 38788, 38801, 38811, 38831,
 38848, 38871, 38880, 38892, 38893,
 39165, 39179, 39193, 39200, 39207,
 39216, 39225, 39236, 39245, 39254,
 39259, 39264, 39272, 39288, 39307,
 39322, 39327, 39332, 39347, 39348,
 39353, 39358, 39363, 39368, 39374,
 39379, 39385, 39399, 39420, 39437,
 39447, 39461, 39496, 39506, 39511,
 39522, 39524, 39554, 39557, 39559,
 39561, 39579, 39616, 39622, 39639,
 39660, 39673, 39686, 39706, 39719,
 39734, 39748, 39757, 39767, 39779,
 39795, 39808, 39844, 39849, 39864,
 39870, 39881, 39894, 39909, 39947,
 39960, 39999, 40005, 40007, 40012,
 40017, 40033, 40038, 40043, 40048,
 40053, 40182, 40195, 40208, 40218,
 40223, 40232, 40237, 40242, 40247,
 40249, 40251, 40461, 40475, 40491,
 40497, 40500, 40507, 40510, 40516,
 40533, 40555, 40564, 40577, 40596,
 40611, 40625, 40632, 40652, 40677,
 40691, 40692, 40693, 40701, 40733,
 40744, 40783, 40822, 40827, 40832,
 40837, 40839, 40841, 40843, 40845,
 40847, 40849, 40861, 40884, 40893,
 40898, 40951, 40972, 40978, 41016,
 41025, 41062, 41105, 41107, 41112,
 41124, 41126, 41128, 41176, 41185,
 41190, 41200, 41206, 41212, 41219,
 41235, 41240, 41247, 41252, 41259,
 41300, 41303, 41325, 41341, 41353,
 41373, 41375, 41387, 41406, 41413,
 41435, 41438, 41440, 41442, 41444,
 41446, 41448, 41450, 41452, 41455,
 41458, 41461, 41465, 41467, 41471,
 41477, 41496, 41498, 41500, 41502,
 41504, 41506, 41529, 41533, 41539,
 41541, 41544, 41546, 41593, 41595,
 41597, 41669, 41671, 41703, 41714,
 41741, 41780, 41787, 41788, 41791,
 41792, 41793, 41794, 41842, 41851,
 41853, 41865, 41878, 41883, 41885,
 41886, 41896, 41913, 41915, 41920
 \cs_new_protected:Npx
 16, 1984, 2003, 2033, 2041, 2131, 2208
 \cs_new_protected_nopar:Nn
 19, 2129, 2206
 \cs_new_protected_nopar:Npe
 17, 1984, 1999, 2024, 2031
 \cs_new_protected_nopar:Npn
 17, 1984, 1998, 2005, 2024, 2030
 \cs_new_protected_nopar:Npx
 17, 1984, 2000, 2024, 2032
 \cs_parameter_spec:N
 24, 2331, 2360, 2368, 13521,
 13556, 41431, 41432, 42088, 42503
 \cs_prefix_spec:N
 24, 2331, 2340, 2348,
 13521, 13556, 42028, 42086, 42502
 \cs_replacement_spec:N
 25, 2331, 2380,
 2388, 3173, 3174, 23678, 32967, 42091
 \cs_set:Nn 20, 415, 2129, 2206
 .cs_set:Np 248, 22980
 \cs_set:Npe 17, 1466,
 1471, 1473, 2015, 2016, 6355, 10748,

- 10749, 10750, 10751, 10752, 12708,
14898, 14926, 19518, 20558, 20575,
20615, 20621, 30986, 41208, 41214
- \cs_set:Npn 15, 17, 66, 401, 411, 415,
967, 1466, 1469, 1596, 1617, 1984,
2004, 2015, 2015, 2129, 2206, 3282,
4653, 4654, 4655, 4984, 4985, 5561,
5563, 5580, 5582, 5822, 5823, 6087,
6088, 6089, 6090, 6116, 6703, 7008,
9074, 9341, 9343, 10690, 11013,
12554, 12706, 12863, 13783, 15544,
15701, 17310, 17332, 19442, 19531,
20024, 20560, 20574, 21042, 22440,
22441, 22981, 22983, 23588, 24644,
24652, 24663, 24680, 24689, 24720,
30840, 31059, 32264, 32266, 32510,
32516, 41411, 41743, 41751, 41759,
41769, 41782, 41783, 41784, 41785,
41814, 41846, 41888, 41897, 42043
- \cs_set:Npx
..... 17, 424, 1466, 1473, 2015, 2017
- \cs_set_eq:NN
..... 21, 67, 412, 604, 961, 964,
970, 1740, 2042, 2042, 2043, 2044,
2045, 2046, 2053, 2779, 2797, 2947,
3545, 3546, 3550, 3751, 3794, 3819,
4193, 4233, 4510, 6079, 6113, 6709,
6758, 7602, 7630, 7930, 7968, 7969,
7971, 7972, 7973, 7994, 8377, 8379,
8755, 10754, 10755, 10756, 10757,
10759, 10761, 10762, 12224, 12226,
14635, 14644, 15446, 17692, 17693,
17695, 17841, 17842, 17853, 19106,
19981, 20184, 20554, 20613, 20620,
20833, 20877, 20941, 20962, 20969,
21013, 21178, 21192, 21208, 22043,
22051, 22506, 22742, 22750, 22830,
30877, 30878, 30894, 30912, 31027,
31038, 31117, 31118, 31691, 32162,
32174, 32188, 41223, 41729, 41737
- \cs_set_nopar:Nn 20, 2129, 2206
- \cs_set_nopar:Npe
..... 17, 482, 754, 961, 963, 964,
970, 973, 1019, 1023, 1038, 1466,
1467, 2004, 2007, 2468, 2655, 4154,
12244, 13585, 13601, 13632, 20833,
20877, 20941, 20942, 20962, 20969,
20992, 21013, 21178, 21192, 21208,
21222, 21226, 22567, 22585, 22727,
23306, 23311, 31050, 32195, 42134
- \cs_set_nopar:Npn
16, 17, 206, 411, 1566, 1466, 1466,
2004, 2006, 21522, 22673, 23327,
23328, 30915, 32135, 32136, 32137,
32138, 32142, 32143, 32144, 32148
- \cs_set_nopar:Npx
17, 1466, 1468, 1489, 2004, 2008, 2150
- \cs_set_protected:Nn . 20, 2129, 2206
- .cs_set_protected:Np 248, 22980
- \cs_set_protected:Npe 17, 110, 1466,
1481, 1483, 2033, 2034, 9707, 22711
- \cs_set_protected:Npn 16, 17, 412,
123, 1466, 1479, 1602, 1623, 2033,
2033, 2965, 3122, 3547, 4107, 5298,
5335, 6064, 6073, 6075, 6077, 6080,
6082, 6091, 6093, 6098, 6100, 6105,
6107, 6109, 6111, 6114, 6867, 6868,
7391, 9460, 9525, 10208, 10812,
10893, 10993, 11009, 12596, 12744,
12870, 13095, 13295, 13692, 15044,
15097, 17515, 17758, 19668, 19970,
20056, 20269, 20288, 20696, 21929,
22112, 22226, 22355, 22397, 22708,
22985, 22987, 23523, 24597, 25112,
25188, 25769, 25843, 25857, 26079,
26096, 26131, 26167, 26182, 26199,
27842, 30681, 30713, 32145, 32189,
32191, 32193, 32201, 32216, 32232,
32242, 32251, 32272, 32300, 32307,
32322, 32333, 32339, 32340, 32341,
32354, 32376, 32410, 32419, 32435,
32442, 32492, 32521, 32538, 32546,
32956, 34868, 34901, 35749, 35802,
35828, 37514, 37527, 37558, 39540,
41404, 41410, 41796, 41805, 41824,
41829, 41835, 41844, 41845, 41847,
41848, 41849, 41880, 41884, 42002,
42008, 42022, 42039, 42064, 42070,
42079, 42490, 42496, 42508, 42569,
42617, 42652, 42676, 42728, 42779
- \cs_set_protected:Npx
..... 17, 1466, 1483, 2033, 2035
- \cs_set_protected_nopar:Nn
..... 20, 2129, 2206
- \cs_set_protected_nopar:Npe
..... 18, 1466, 1476, 1478, 2024, 2025
- \cs_set_protected_nopar:Npn
..... 18, 411, 1466, 1474, 2024, 2024
- \cs_set_protected_nopar:Npx
..... 18, 1466, 1478, 2024, 2026
- \cs_show:N
.. 22, 29, 419, 2295, 2295, 2296, 2297
- \cs_split_function:N ... 24, 1612,
1633, 1750, 1751, 1808, 1810, 2101,
2142, 2766, 3055, 17173, 17194, 41923
- \cs_to_str:N
... 6, 23, 118, 134, 405, 406, 756,
777, 1799, 1799, 1814, 4396, 5894,

- 6030, 10675, 13589, 13651, 13656,
13664, 14417, 14418, 14419, 14420,
14421, 14422, 14423, 14424, 14425,
14426, 14427, 14428, 17315, 17337,
19091, 24595, 31308, 31314, 31316,
31324, 31387, 31391, 31398, 31445,
31450, 31486, 33235, 35737, 41852,
42015, 42024, 42033, 42047, 42056
- \cs_undefine:N
..... 22, 905, 1018, 1025, 2058,
2058, 2060, 2066, 9576, 9577, 9578,
10306, 10549, 11282, 11835, 11836,
13322, 13580, 30893, 30973, 30974,
31144, 31649, 31710, 32422, 32433
- cs internal commands:
- __cs_count_signature:N
..... 404, 2100, 2100, 2112, 2113
- __cs_count_signature:n
..... 2100, 2101, 2102
- __cs_count_signature:nnN
..... 2100, 2103, 2104
- __cs_generate_from_signature:n .
..... 2151, 2165
- __cs_generate_from_signature:NNn
..... 2133, 2137
- __cs_generate_from_signature:nnNNNn
..... 2141, 2146
- __cs_generate_internal_c:NN . 3018
- __cs_generate_internal_end:w ...
..... 3001, 3035
- __cs_generate_internal_long:nnnNNn
..... 3039, 3043
- __cs_generate_internal_long:w ...
..... 3002, 3037
- __cs_generate_internal_loop:nwnnw
..... 2999,
3005, 3017, 3019, 3021, 3023, 3026
- __cs_generate_internal_N:NN . 3016
- __cs_generate_internal_n:NN . 3020
- __cs_generate_internal_one_-
go:NNn 441, 2988, 2997
- __cs_generate_internal_other:NN
..... 3010, 3024
- __cs_generate_internal_test:Nw .
..... 2973, 2993
- __cs_generate_internal_test_-
aux:w 2975, 2991, 2994
- __cs_generate_internal_variant:n
... 444, 2938, 2943, 2943, 3119, 3125
- __cs_generate_internal_variant:NNn
..... 441, 2963, 2967
- __cs_generate_internal_variant:wnnNwn
..... 2945, 2958
- __cs_generate_internal_variant_-
loop:n . 2943, 2983, 3040, 3045, 3048
- __cs_generate_internal_x:NN . 3022
- __cs_generate_variant:N
..... 2762, 2775, 2775
- __cs_generate_variant:n 3050
- __cs_generate_variant:nnNN
..... 2765, 2798, 2798
- __cs_generate_variant:nnNnn
..... 3050, 3054, 3058
- __cs_generate_variant:Nnnw
..... 2805, 2807, 2807, 2825
- __cs_generate_variant:w
..... 3050, 3065, 3069, 3086
- __cs_generate_variant:ww
..... 2775, 2781, 2791
- __cs_generate_variant:wwNN
... 437, 438, 2814, 2931, 2931, 42378
- __cs_generate_variant:wwNw
..... 2775, 2793, 2795
- __cs_generate_variant_check:nn .
.. 3050, 3089, 3091, 3093, 3095, 3096
- __cs_generate_variant_F_-
form:nnn 3050, 3092
- __cs_generate_variant_loop:nNwN
... 437, 438, 2815, 2827, 2827, 2846
- __cs_generate_variant_loop_-
base:N 2827, 2832, 2835, 2848
- __cs_generate_variant_loop_-
end:nwwwNNnn
..... 437, 438, 2817, 2827, 2873
- __cs_generate_variant_loop_-
invalid:NNwNNnn
..... 437, 2827, 2839, 2894
- __cs_generate_variant_loop_-
long:wNNnn .. 438, 2820, 2827, 2881
- __cs_generate_variant_loop_-
same:w 437, 2827, 2830, 2870
- __cs_generate_variant_loop_-
special:NNwNNnn
..... 2827, 2837, 2909, 2926
- __cs_generate_variant_p_-
form:nnn 3050, 3088
- __cs_generate_variant_same:N ...
..... 437, 2872, 2920, 2920
- __cs_generate_variant_T_-
form:nnn 3050, 3090
- __cs_generate_variant_TF_-
form:nnn 3050, 3094
- __cs_if_exist_c_aux: 1824, 1841, 1847
- __cs_if_exist_c_aux:w
..... 1824, 1854, 1863
- __cs_if_exist_use_aux:Nnn
..... 1910, 1928, 1939, 1941

- __cs_if_exist_use_aux:w
..... [1910](#), [1923](#), [1927](#), [1934](#), [1938](#)
 - __cs_if_free_c_aux:w
..... [1883](#), [1891](#), [1898](#), [1907](#)
 - __cs_macro_parameter_spec:N ...
..... [2331](#), [2358](#), [2363](#)
 - __cs_macro_prefix_spec:N
..... [2331](#), [2338](#), [2343](#)
 - __cs_macro_replacement_spec:N ..
..... [2331](#), [2378](#), [2383](#)
 - __cs_parm_from_arg_count_-
test:nnTF [2069](#), [2071](#), [2090](#)
 - __cs_prefix_arg_replacement:wN .
..... [2331](#), [2333](#), [2352](#), [2372](#), [2392](#)
 - __cs_split_function_auxi:w
..... [1808](#), [1813](#), [1817](#)
 - __cs_split_function_auxii:w ...
..... [1808](#), [1819](#), [1820](#)
 - __cs_tmp:w [404](#), [435](#), [440](#), [444](#), [1808](#),
[1823](#), [1984](#), [1984](#), [1992](#), [1993](#), [1994](#),
[1995](#), [1996](#), [1997](#), [1998](#), [1999](#), [2000](#),
[2001](#), [2002](#), [2003](#), [2004](#), [2006](#), [2007](#),
[2008](#), [2009](#), [2010](#), [2011](#), [2012](#), [2013](#),
[2014](#), [2015](#), [2016](#), [2017](#), [2018](#), [2019](#),
[2020](#), [2021](#), [2022](#), [2023](#), [2024](#), [2025](#),
[2026](#), [2027](#), [2028](#), [2029](#), [2030](#), [2031](#),
[2032](#), [2033](#), [2034](#), [2035](#), [2036](#), [2037](#),
[2038](#), [2039](#), [2040](#), [2041](#), [2129](#), [2150](#),
[2152](#), [2155](#), [2170](#), [2171](#), [2172](#), [2173](#),
[2174](#), [2175](#), [2176](#), [2177](#), [2178](#), [2179](#),
[2180](#), [2181](#), [2182](#), [2183](#), [2184](#), [2185](#),
[2186](#), [2187](#), [2188](#), [2189](#), [2190](#), [2191](#),
[2192](#), [2193](#), [2194](#), [2195](#), [2196](#), [2197](#),
[2198](#), [2199](#), [2200](#), [2201](#), [2202](#), [2203](#),
[2204](#), [2205](#), [2206](#), [2214](#), [2215](#), [2216](#),
[2217](#), [2218](#), [2219](#), [2220](#), [2221](#), [2222](#),
[2223](#), [2224](#), [2225](#), [2226](#), [2227](#), [2228](#),
[2229](#), [2230](#), [2231](#), [2232](#), [2233](#), [2234](#),
[2235](#), [2236](#), [2237](#), [2238](#), [2239](#), [2240](#),
[2241](#), [2242](#), [2243](#), [2244](#), [2245](#), [2246](#),
[2247](#), [2248](#), [2249](#), [2779](#), [2797](#), [2939](#),
[2947](#), [2965](#), [2996](#), [3122](#), [3129](#), [3130](#),
[3131](#), [3132](#), [3133](#), [3134](#), [3135](#), [3136](#),
[3137](#), [3138](#), [3139](#), [3140](#), [3141](#), [3142](#),
[3143](#), [3144](#), [3145](#), [3146](#), [3147](#), [3148](#),
[3149](#), [3150](#), [3151](#), [3152](#), [3153](#), [3154](#),
[3155](#), [3156](#), [3157](#), [3158](#), [3159](#), [3160](#),
[3161](#), [3162](#), [3163](#), [3164](#), [3165](#), [3166](#),
[3167](#), [3168](#), [3169](#), [3170](#), [3171](#), [3172](#)
 - __cs_to_str:N
..... [405](#), [1799](#), [1803](#), [1805](#), [1806](#)
 - __cs_to_str:w . [405](#), [1799](#), [1802](#), [1806](#)
 - __cs_use_i_delimit_by_s_stop:nw
..... [2756](#), [2757](#), [3063](#)
 - __cs_use_none_delimit_by_q_-
recursion_stop:w
..... [2756](#), [2758](#), [2803](#), [2810](#), [3076](#)
 - __cs_use_none_delimit_by_s_-
stop:w [2756](#), [2756](#), [3067](#)
 - csc [283](#)
 - cscd [284](#)
 - \csname [682](#),
[4](#), [8](#), [13](#), [17](#), [30](#), [53](#), [54](#), [61](#), [94](#), [184](#)
 - \csstring [804](#)
 - \currentcjktoken [1136](#)
 - \currentgrouplevel [476](#)
 - \currentgrouptype [477](#)
 - \currentifbranch [478](#)
 - \currentiflevel [479](#)
 - \currentiftype [480](#)
 - \currentspacingmode [1137](#)
 - \currentxspacingmode [1138](#)
- D**
- \d [33333](#), [35799](#), [35821](#)
 - \date [383](#)
 - \day [185](#), [1287](#), [9098](#)
 - dd [287](#)
 - \deadcycles [186](#)
 - debug commands:
 - \debug_assert:nN
..... [32](#), [1577](#), [1577](#), [41741](#), [41743](#), [41782](#)
 - \debug_assert:nn [31](#),
[32](#), [1577](#), [1578](#), [41741](#), [41751](#), [41783](#)
 - \debug_assert:nNn . [32](#), [1577](#), [1579](#),
[3173](#), [3175](#), [41741](#), [41746](#), [41759](#), [41784](#)
 - \debug_assert:nnn . [32](#), [1577](#), [1580](#),
[3173](#), [3176](#), [41741](#), [41754](#), [41769](#), [41785](#)
 - \debug_off:n . [31](#), [383](#), [1552](#), [1553](#),
[1563](#), [1569](#), [1573](#), [41687](#), [41695](#), [41716](#)
 - \debug_on:n [31](#), [32](#), [383](#), [714](#), [1552](#),
[1563](#), [1563](#), [1567](#), [41687](#), [41687](#), [41705](#)
 - \debug_resume: [31](#), [1307](#), [1449](#), [1563](#),
[1575](#), [1576](#), [31535](#), [37433](#), [41725](#), [41731](#)
 - \debug_suspend:
..... [31](#), [1307](#), [1449](#), [1563](#),
[1575](#), [1575](#), [31533](#), [37426](#), [41725](#), [41726](#)
 - debug internal commands:
 - __debug_add_to_debug_code:Nnn ..
..... [42001](#), [42016](#), [42039](#)
 - __debug_all_off: [41687](#), [41714](#)
 - __debug_all_on:
..... [1566](#), [1572](#), [41687](#), [41703](#)
 - __debug_arg_check_invalid:N ...
..... [41920](#), [41942](#), [41948](#)
 - __debug_arg_if_braced:N [41963](#)
 - __debug_arg_if_braced:n
..... [41920](#), [41964](#), [41965](#)

- __debug_arg_if_braced:NTF 41761, 41771, 41798, 41807, 41816, 41826, 41831, 41837, 41881, 41890
- __debug_arg_list_from_signature:nNN .. 41920, 41931, 41936, 41939, 41945
- __debug_arg_return:N 41920, 41978, 41979, 41980, 41981, 41982, 41983, 41984, 41985, 41986, 41987, 41999
- __debug_build_arg_list:n 41920, 41927, 41934
- __debug_build_parm_text:n 41920, 41925, 41929
- __debug_check-assertions_off: 41741
- __debug_check-assertions_on: . 41741
- __debug_check-declarations_off: 41787
- __debug_check-declarations_on: 41787
- __debug_check-expressions_off: 41886
- __debug_check-expressions_on: 41886
- __debug_chk_expr_aux:nNnN 41886, 41891, 41899
- __debug_chk_var_scope_aux:NN ... 41827, 41833, 41839, 41851, 41851
- __debug_chk_var_scope_aux:Nn ... 41851, 41852, 41853
- __debug_chk_var_scope_aux:NnN .. 1566, 41851, 41856, 41860, 41865
- __debug_deprecation_off: 41913, 41915
- \g__debug_deprecation_off_tl ... 1581, 41916
- __debug_deprecation_on: 41913, 41913
- \g__debug_deprecation_on_tl 1581, 41914
- __debug_generate_parameter_-list:NNN 1568, 1570, 41920, 41920, 42025
- __debug_get_base_form:N 41920, 41964, 41976
- __debug_if_recursion_tail_-stop:N 41684, 41686, 41941
- __debug_insert_debug_code:Nnn 42001
- __debug_log-functions_off: .. 41878
- __debug_log-functions_on: ... 41878
- __debug_parm_terminate:w 41920, 41950, 41951, 41952, 41953, 41961
- __debug_patch_weird:Nnn 42001, 42076, 42079
- __debug_setup_debug_code:Nnn ... 42001, 42017, 42022
- __debug_suspended:TF 1563, 41725, 41729, 41737, 41740, 41745, 41753, 41761, 41771, 41798, 41807, 41816, 41826, 41831, 41837, 41881, 41890
- \l__debug_suspended_tl 41725
- __debug_tmp:w 42043, 42062
- \l__debug_tmp_tl 41917, 41922, 41925, 41927
- \l__debug_tmpa_tl 41917, 42025, 42030
- \l__debug_tmpp_tl 41917, 42025, 42034
- __debug_use_i_delimit_by_s_-stop:nw . 41681, 41681, 41855, 41857
- __debug_use_none_delimit_by_q_-recursion_stop:w ... 41684, 41962
- decodearray 339
- \def 36, 37, 38, 60, 62, 67, 68, 70, 84, 106, 143, 187
- default commands:
 - .default:n 248, 22996
- \defaulthyphenchar 188
- \defaultskewchar 189
- \deferred 805
- deg 286
- \delcode 190
- \delimiter 191
- \delimiterfactor 192
- \delimitershortfall 193
- deprecation internal commands:
 - __deprecation_just_error:nnNN 41373
 - __deprecation_patch_aux:Nn 41373, 41385, 41406
 - __deprecation_patch_aux:nnNNnn 41373, 41374, 41375
 - __deprecation_warn_once:nnNnn .. 41373, 41384, 41387
- \detokenize 30, 94, 481
- \DH 33339, 34880, 35762
- \dh 33339, 34880, 35772
- dim commands:
 - \dim_abs:n 230, 992, 21644, 21644, 42593
 - \dim_add:Nn 230, 21625, 21625, 21632, 42152, 42528
 - \dim_case:nn 233, 21748, 21763
 - \dim_case:nnTF 233, 21748, 21748, 21753, 21758
 - \dim_compare:n 21708
 - \dim_compare:nNn 21679
 - \dim_compare:nNnTF 231-234, 270, 21679, 21772, 21808, 21816, 21825, 21831, 21843, 21846, 21857, 22011, 37260, 37277, 37311, 37325, 37335, 37800, 37803, 37808, 37822, 37825, 37830, 38176, 38181, 38191, 38326, 38338, 40466, 40468
 - \dim_compare:nTF 231, 232, 234, 21708, 21780, 21788, 21797, 21803

- \dim_compare_p:n [232](#), [21708](#)
- \dim_compare_p:nNn [231](#),
[21679](#), [21684](#), [21686](#), [21688](#), [41147](#),
[41148](#), [41155](#), [41156](#), [41159](#), [41160](#)
- \dim_const:Nn
... [229](#), [985](#), [1000](#), [21592](#), [21592](#),
[21597](#), [22032](#), [22033](#), [24009](#), [40467](#),
[40469](#), [40470](#), [40471](#), [42305](#), [42532](#)
- \dim_do_until:nn
..... [234](#), [21778](#), [21800](#), [21804](#)
- \dim_do_until:nNnn
..... [233](#), [21806](#), [21828](#), [21832](#)
- \dim_do_while:nn
..... [234](#), [21778](#), [21794](#), [21798](#)
- \dim_do_while:nNnn
..... [233](#), [21806](#), [21822](#), [21826](#)
- \dim_eval:n [231](#),
[232](#), [235](#), [985](#), [1421](#), [1422](#), [21595](#),
[21751](#), [21756](#), [21761](#), [21766](#), [21861](#),
[21889](#), [21889](#), [22027](#), [22031](#), [37170](#),
[37176](#), [37212](#), [37219](#), [37490](#), [37567](#),
[37661](#), [37679](#), [37720](#), [37724](#), [37725](#),
[37729](#), [37733](#), [37734](#), [37751](#), [37756](#),
[37762](#), [37769](#), [37776](#), [37942](#), [37945](#),
[37946](#), [37953](#), [38035](#), [38036](#), [38043](#),
[38044](#), [38147](#), [38154](#), [38303](#), [38304](#),
[38575](#), [38576](#), [38577](#), [40662](#), [42590](#)
- \dim_gadd:Nn
[230](#), [21625](#), [21627](#), [21633](#), [42234](#), [42529](#)
- .dim_gset:N [248](#), [23006](#)
- \dim_gset:Nn [230](#),
[985](#), [21613](#), [21615](#), [21618](#), [42233](#), [42527](#)
- \dim_gset_eq:NN
.... [230](#), [21619](#), [21622](#), [21624](#), [42231](#)
- \dim_gsub:Nn
[230](#), [21625](#), [21636](#), [21642](#), [42235](#), [42531](#)
- \dim_gzero:N [229](#), [21598](#),
[21599](#), [21602](#), [21606](#), [22070](#), [42232](#)
- \dim_gzero_new:N
..... [229](#), [21603](#), [21605](#), [21608](#)
- \dim_horizontal:N
.... [238](#), [22038](#), [22043](#), [22044](#), [37144](#)
- \dim_horizontal:n [238](#), [22038](#),
[22038](#), [22043](#), [36853](#), [37153](#), [37184](#),
[37186](#), [37195](#), [37228](#), [37233](#), [37256](#),
[37258](#), [37307](#), [37309](#), [37919](#), [38177](#),
[38182](#), [38264](#), [38265](#), [38550](#), [38551](#)
- \dim_if_exist:N [21609](#), [21611](#)
- \dim_if_exist:NTF [230](#), [21604](#), [21606](#),
[21609](#), [40463](#), [40477](#), [40481](#), [40484](#)
- \dim_if_exist_p:N [230](#), [21609](#)
- \dim_log:N ... [237](#), [22028](#), [22028](#), [22029](#)
- \dim_log:n [237](#), [22028](#), [22030](#)
- \dim_max:nn
[230](#), [21644](#), [21651](#), [38014](#), [38018](#), [42639](#)
- \dim_min:nn [230](#), [21644](#),
[21659](#), [38012](#), [38016](#), [38029](#), [42640](#)
- \dim_new:N [229](#), [21586](#), [21586](#),
[21591](#), [21594](#), [21604](#), [21606](#), [22034](#),
[22035](#), [22036](#), [22037](#), [36630](#), [36631](#),
[36808](#), [36809](#), [36810](#), [36811](#), [36812](#),
[36813](#), [36814](#), [36815](#), [37353](#), [37377](#),
[37378](#), [37379](#), [37380](#), [37381](#), [37382](#),
[37888](#), [37889](#), [37890](#), [37891](#), [37892](#),
[38050](#), [38051](#), [38398](#), [38400](#), [38401](#),
[40429](#), [40457](#), [40458](#), [40459](#), [40460](#)
- \dim_ratio:nn [231](#), [21675](#), [21675](#)
- .dim_set:N [248](#), [23006](#)
- \dim_set:Nn [230](#), [21613](#), [21613](#),
[21617](#), [36640](#), [36661](#), [36701](#), [36835](#),
[36836](#), [36837](#), [36869](#), [36880](#), [36964](#),
[36965](#), [36966](#), [36981](#), [37079](#), [37080](#),
[37081](#), [37083](#), [37085](#), [37087](#), [37480](#),
[37556](#), [37802](#), [37806](#), [37824](#), [37828](#),
[37859](#), [37873](#), [37948](#), [37983](#), [37991](#),
[38002](#), [38003](#), [38004](#), [38005](#), [38011](#),
[38013](#), [38015](#), [38017](#), [38022](#), [38028](#),
[38111](#), [38113](#), [38115](#), [38123](#), [38125](#),
[38179](#), [38254](#), [38255](#), [38257](#), [38259](#),
[38277](#), [38278](#), [38399](#), [38494](#), [38495](#),
[38541](#), [38542](#), [38543](#), [38545](#), [40579](#),
[40580](#), [40581](#), [40582](#), [42150](#), [42526](#)
- \dim_set_eq:NN [230](#),
[21619](#), [21619](#), [21621](#), [37505](#), [37506](#),
[40479](#), [40480](#), [40482](#), [40485](#), [42151](#)
- \dim_show:N .. [237](#), [22024](#), [22024](#), [22025](#)
- \dim_show:n ... [237](#), [998](#), [22026](#), [22026](#)
- \dim_sign:n .. [235](#), [21891](#), [21891](#), [42594](#)
- \dim_step_function:nnnN
..... [234](#), [992](#), [21834](#), [21834](#),
[21886](#), [42694](#), [42698](#), [42702](#), [42706](#)
- \dim_step_inline:nnnn
..... [235](#), [21864](#), [21864](#)
- \dim_step_variable:nnnN
..... [235](#), [21864](#), [21871](#)
- \dim_sub:Nn
[230](#), [21625](#), [21634](#), [21641](#), [42153](#), [42530](#)
- \dim_to_decimal:n
..... [235](#), [996](#), [21911](#), [21911](#),
[21947](#), [21975](#), [22012](#), [22021](#), [42591](#)
- \dim_to_decimal_in_bp:n .. [236](#), [21928](#)
- \dim_to_decimal_in_cc:n .. [236](#), [21928](#)
- \dim_to_decimal_in_cm:n .. [236](#), [21928](#)
- \dim_to_decimal_in_dd:n .. [236](#), [21928](#)
- \dim_to_decimal_in_in:n .. [236](#), [21928](#)
- \dim_to_decimal_in_mm:n .. [236](#), [21928](#)
- \dim_to_decimal_in_pc:n .. [236](#), [21928](#)

- \dim_to_decimal_in_sp:n
 237, 1108, 21926,
 21926, 25250, 25287, 25882, 42592
- \dim_to_decimal_in_unit:nn
 237, 21954, 21954
- \dim_to_fp:n 237, 1108, 1127,
 21926, 30231, 30231, 36873, 36874,
 36884, 36885, 36953, 36956, 36957,
 36982, 36997, 36998, 37017, 37018,
 37036, 37053, 37056, 37057, 37113,
 37115, 37813, 37814, 37815, 37835,
 37836, 37837, 37847, 37848, 37864,
 37865, 37866, 37867, 37877, 37878,
 37987, 37988, 37995, 37996, 38069,
 38072, 38073, 38124, 38126, 42725
- \dim_until_do:nn
 234, 21778, 21786, 21791
- \dim_until_do:nNnn
 234, 21806, 21814, 21819
- \dim_use:N 235,
 1421, 21647, 21653, 21654, 21655,
 21661, 21662, 21663, 21711, 21730,
 21890, 21894, 21909, 21909, 21910,
 21914, 22125, 22126, 22201, 22202,
 37950, 37954, 37961, 37967, 37976,
 37977, 37978, 38132, 38139, 38285
- \dim_vertical:N
 238, 22038, 22051, 22052
- \dim_vertical:n
 238, 22038, 22045, 22051, 37198
- \dim_while_do:nn
 234, 21778, 21778, 21783
- \dim_while_do:nNnn
 234, 21806, 21806, 21811
- \dim_zero:N 229, 21598, 21598, 21601,
 21604, 22069, 36838, 36967, 37082,
 37793, 37794, 40483, 40486, 42149
- \dim_zero_new:N
 229, 21603, 21603, 21607
- \c_max_dim ... 236, 238, 241, 1057,
 22032, 22144, 24036, 24078, 24086,
 38002, 38003, 38004, 38005, 38022
- \g_tmpa_dim 238, 22034
- \l_tmpa_dim 238, 22034
- \g_tmpb_dim 238, 22034
- \l_tmpb_dim 238, 22034
- \c_zero_dim
 238, 21843, 21846, 21899, 22032,
 22143, 24103, 36603, 36635, 36637,
 36651, 36653, 37191, 37203, 37234,
 37264, 37275, 37281, 37293, 37311,
 37315, 37323, 37325, 37329, 37335,
 37345, 37800, 37803, 37808, 37822,
 37825, 37830, 38176, 38181, 38191
- dim internal commands:
 __dim_abs:N 21644, 21646, 21649
 __dim_branch_unit:w
 996, 21964, 21969, 21969
 __dim_case:nnTF 21748,
 21751, 21756, 21761, 21766, 21768
 __dim_case:nw
 21748, 21769, 21770, 21774
 __dim_case_end:nw 21748, 21773, 21776
 __dim_chk_unit:w
 995, 21956, 21959, 21959
 __dim_compare:w . 21708, 21710, 21713
 __dim_compare:wNn
 988, 21708, 21716, 21719, 21729
 __dim_compare_!:w 21708
 __dim_compare_<:w 21708
 __dim_compare_=:w 21708
 __dim_compare_>:w 21708
 __dim_compare_end:w .. 21716, 21740
 __dim_compare_error:
 988, 21708, 21711, 21713, 21742, 21746
 __dim_convert_remainder:w
 997, 21999, 22003, 22003
 __dim_eval:w 21580, 21581, 21614,
 21616, 21626, 21630, 21635, 21639,
 21647, 21653, 21654, 21655, 21661,
 21662, 21663, 21678, 21681, 21711,
 21730, 21735, 21837, 21838, 21839,
 21890, 21894, 21914, 21927, 21934,
 21957, 21965, 22012, 42534, 42596,
 42642, 42673, 42697, 42701, 42705
 __dim_eval_end:
 21580, 21582, 21614, 21616, 21626,
 21630, 21635, 21639, 21647, 21657,
 21665, 21678, 21681, 21890, 21894,
 21914, 21927, 21934, 21957, 22012
 __dim_get_quotient:w
 997, 21976, 21979, 21979
 __dim_get_remainder:w
 997, 21986, 21991, 21997, 21997
 __dim_kern:n
 22038, 22041, 22049, 22053
 __dim_maxmin:wwN
 21644, 21653, 21661, 21667
 __dim_parse_decimal:w
 998, 22013, 22015, 22018, 22018
 __dim_parse_decimal_aux:w
 998, 22018, 22020, 22023
 __dim_ratio:n .. 21675, 21676, 21677
 __dim_sep: 995-
 998, 21643, 21643, 21654, 21655,
 21662, 21663, 21667, 21837, 21838,
 21839, 21841, 21894, 21897, 21934,
 21945, 21957, 21959, 21965, 21966,

21969, 21972, 21976, 21979, 21987, 21988, 21992, 21993, 21997, 22000, 22001, 22003, 22006, 22007, 22009, 22013, 22015, 22018, 22021, 22023	\efcode 670
__dim_sign:Nw . . . 21891, 21893, 21897	\elapsedtime 769
__dim_step:NnnN	\else 9, 11, 18, 54, 55, 56, 207
.. 21834, 21844, 21851, 21855, 21860	else commands:
__dim_step:NNnnnn	\else: 29,
.. 21864, 21867, 21874, 21883	67, 74, 102, 185, 186, 244, 323, 398,
__dim_step:wwwN . 21834, 21836, 21841	400, 406, 435, 610, 742, 944, 1155,
__dim_test_candidate:w	1386, 1389, 1431, 1663, 1671, 1697,
.. 997, 22005, 22009, 22009	1828, 1832, 1843, 1848, 1859, 1864,
__dim_tmp:w . 21929, 21937, 21938,	1869, 1874, 1887, 1903, 2063, 2085,
21939, 21940, 21941, 21942, 21943	2094, 2108, 2167, 2168, 2253, 2490,
__dim_to_decimal:w	2746, 2780, 2831, 2832, 2834, 2838,
.. 21911, 21914, 21918	2850, 2851, 2852, 2853, 2854, 2855,
__dim_to_decimal_aux:w	2856, 2857, 2858, 2922, 2923, 2925,
995, 996, 21928, 21933, 21945, 21972	2974, 3077, 3227, 3228, 3729, 3732,
__dim_use_none_delimit_by_s_-	3735, 3745, 3760, 3787, 3802, 3829,
stop:w 21585, 21585, 21726	3845, 3879, 3887, 3889, 3891, 3893,
\dimen 194, 20330	3895, 3897, 3899, 3901, 3924, 3945,
\dimendef 195	3949, 4032, 4036, 4148, 4171, 4182,
\dimexpr 482	4215, 4217, 4241, 4280, 4291, 4324,
\directlua 21, 23, 808	4500, 4501, 4505, 4506, 4522, 4529,
\disablecjktoken 1200	4729, 4739, 4789, 4798, 4811, 4812,
\discretionary 196	4814, 4816, 4819, 4820, 4824, 4829,
\discretionaryligaturemode 806	4840, 4844, 4848, 4855, 4920, 4927,
\disinhibitglue 1139	4937, 4939, 4949, 4956, 4958, 4969,
\displayindent 197	5089, 5203, 5246, 5251, 5263, 5268,
\displaylimits 198	5358, 5504, 5517, 5606, 5635, 5674,
\displaystyle 199	5692, 5805, 5861, 5895, 5925, 6347,
\displaywidowpenalties 483	6365, 6384, 6418, 6471, 6518, 6522,
\displaywidowpenalty 200	6529, 6550, 6561, 6708, 6820, 6930,
\displaywidth 201	6973, 6976, 7096, 7107, 7116, 7144,
\divide 202	7156, 7182, 7199, 7207, 7476, 7730,
\DJ 33340, 34881, 35763	8010, 8021, 8419, 8470, 8491, 8513,
\dj 33340, 34881, 35773	8531, 8547, 8557, 8573, 8583, 8698,
\do 1251	8700, 8702, 8704, 10400, 10403,
\DocumentMetadata 342	10406, 11188, 11195, 11458, 11467,
\doublehyphendemerits 203	11478, 12219, 12780, 12790, 12805,
\dp 204	12814, 12833, 12847, 12877, 12895,
draft 339	12910, 13179, 13197, 13217, 13225,
\draftmode 934	13235, 13251, 13274, 13285, 13291,
draw commands:	13439, 13451, 13500, 13503, 13506,
\draw_begin: 335	13822, 13827, 13832, 13839, 13844,
\draw_end: 335	14097, 14153, 14156, 14159, 14171,
\dtou 1140	14186, 14327, 14517, 14525, 14533,
\dump 205	14682, 14733, 14734, 14738, 14743,
\dviextension 809	14766, 14819, 14919, 15170, 15200,
\dvifedback 810	15203, 15233, 15236, 15253, 15256,
\dvivariable 811	15360, 15365, 15383, 15402, 15405,
E	15454, 15459, 15462, 15577, 15589,
\edef 73, 82, 206	15598, 15720, 15725, 16634, 16641,
	17078, 17116, 17124, 17135, 17145,
	17164, 17188, 17192, 17234, 17294,
	17305, 17324, 17708, 17778, 17787,
	18225, 18236, 18257, 18273, 18276,

18297, 18338, 18461, 18488, 18520,	29740, 29741, 29742, 29847, 29869,
18558, 18566, 18875, 18908, 18959,	29884, 29902, 29950, 29966, 30175,
19076, 19103, 19130, 19139, 19343,	30242, 30247, 30385, 30421, 30434,
19358, 19380, 19394, 20005, 20011,	30464, 30468, 30476, 30503, 30532,
20014, 20032, 20099, 20102, 20105,	30540, 30557, 30560, 31164, 31168,
20108, 20111, 20114, 20117, 20120,	31220, 31279, 31291, 31371, 32015,
20123, 20126, 20166, 20171, 20176,	32026, 32046, 32502, 32692, 32696,
20181, 20188, 20195, 20200, 20205,	32707, 32722, 32734, 32738, 32747,
20210, 20215, 20220, 20227, 20232,	32748, 32749, 32750, 32751, 32752,
20254, 20260, 20263, 20299, 20302,	32753, 32754, 32755, 32766, 32780,
20346, 20354, 20359, 20465, 20474,	32783, 32786, 32789, 32792, 32795,
20482, 20491, 20567, 20592, 20596,	32798, 32892, 32898, 32907, 32911,
20606, 20644, 20658, 20667, 20677,	33317, 34250, 34254, 34258, 34261,
20711, 21095, 21143, 21152, 21300,	34265, 34279, 34282, 34285, 34288,
21338, 21348, 21650, 21671, 21682,	34291, 34294, 34314, 34317, 34320,
21716, 21741, 21901, 21904, 21950,	34323, 34326, 34329, 34332, 34335,
22503, 24041, 24270, 24287, 24288,	34338, 34341, 34344, 34347, 34350,
24303, 24313, 24408, 24485, 24548,	34353, 34356, 34359, 34362, 34391,
24551, 24565, 24583, 24587, 24843,	34394, 34409, 34412, 34426, 34429,
24856, 24876, 24904, 24905, 24927,	34432, 34435, 34451, 34454, 34457,
24948, 24972, 24973, 25008, 25025,	36478, 36480, 36486, 41859, 41868,
25043, 25078, 25082, 25118, 25135,	41871, 41950, 41951, 41952, 41953,
25141, 25145, 25149, 25308, 25341,	41967, 41968, 41978, 41979, 41980,
25349, 25382, 25386, 25398, 25408,	41981, 41982, 41983, 41984, 41985,
25418, 25449, 25462, 25498, 25508,	41986, 42756, 42757, 42762, 42763
25527, 25540, 25553, 25557, 25568,	em 287
25591, 25608, 25620, 25634, 25647,	\emergencystretch 208
25651, 25659, 25661, 25671, 25682,	\enablecjktoken 1201
25698, 25712, 25718, 25721, 25728,	\end .. 378, 209, 32936, 32946, 35732, 40522
25750, 25780, 25803, 25831, 25834,	\endcsname 682,
26012, 26016, 26023, 26042, 26054,	4, 8, 13, 17, 30, 53, 54, 61, 94, 210
26058, 26065, 26087, 26104, 26110,	\endgroup . 3, 7, 12, 16, 36, 67, 71, 77, 211
26142, 26174, 26190, 26210, 26251,	\endinput 78, 212
26266, 26299, 26301, 26307, 26322,	\endL 484
26375, 26593, 26609, 26620, 26669,	\endlinechar 93, 104, 213
26672, 26675, 26678, 26709, 26718,	\endlocalcontrol 814
26727, 26730, 26897, 26910, 26913,	\endR 485
26920, 26939, 26963, 26964, 26980,	\ensuremath 1348, 32941
26990, 27039, 27042, 27051, 27063,	\epTeXinputencoding 1141
27074, 27088, 27101, 27141, 27177,	\epTeXversion 1142
27198, 27235, 27254, 27257, 27263,	\eqno 214
27277, 27313, 27331, 27334, 27337,	\errhelp 68, 215
27340, 27402, 27478, 27550, 27551,	\errmessage 70, 216
27560, 27595, 27678, 27682, 27686,	\errorcontextlines 68, 217
27748, 27783, 27798, 28074, 28105,	\errorstopmode 218
28109, 28273, 28282, 28345, 28356,	\escapechar 219
28372, 28380, 28441, 28529, 28540,	escapehex 11976
28545, 28579, 28592, 28604, 28610,	\ETC 4346
28731, 28739, 28780, 28787, 28809,	\TeXglueshrinkorder 812
28836, 28851, 28855, 28878, 28909,	\TeXgluestretchorder 813
28912, 28937, 28940, 28982, 28990,	\TeXrevision 486
29001, 29004, 29119, 29134, 29149,	\TeXversion 487
29164, 29179, 29194, 29215, 29260,	\etoksapp 815
29566, 29604, 29605, 29614, 29675,	\etokspre 816

- `\euc` 1143
- `\everycr` 220
- `\everydisplay` 221
- `\everyeof` 488
- `\everyhbox` 222
- `\everyjob` 28, 29, 223
- `\everymath` 224
- `\everypar` 225
- `\everyvbox` 226
- `ex` 287
- `\exceptionpenalty` 817
- `\exhyphenchar` 818
- `\exhyphenpenalty` 227
- `exp` 282
- exp commands:
 - `\exp:w` 44, 45, 398, 405, 426, 427, 434, 592–595, 613, 676, 745, 746, 755, 769, 910, 1098, 1100, 1101, 1104, 1105, 1123, 1128, 1408, 1590, 1592, 2462, 2475, 2481, 2523, 2527, 2532, 2538, 2544, 2556, 2568, 2574, 2580, 2585, 2587, 2594, 2601, 2639, 2644, 2651, 2660, 2662, 2666, 2673, 2679, 2687, 2696, 2703, 2718, 2731, 2735, 2740, 2742, 3030, 7931, 7939, 7977, 8015, 8024, 8035, 8481, 8628, 8630, 8632, 8634, 8651, 8708, 8711, 10858, 12551, 12907, 13361, 13444, 13602, 13608, 13625, 13643, 13864, 13869, 13874, 13879, 13899, 13904, 13909, 13914, 14080, 14089, 14144, 18526, 18531, 18536, 18541, 19197, 19205, 19266, 19568, 19578, 19992, 20497, 20499, 20501, 20503, 20505, 20507, 20509, 20511, 20513, 20515, 20517, 20519, 20586, 21715, 21750, 21755, 21760, 21765, 24316, 24431, 24435, 24819, 24945, 24946, 24947, 24948, 25070, 25088, 25117, 25161, 25173, 25178, 25186, 25194, 25215, 25221, 25293, 25306, 25307, 25316, 25329, 25347, 25348, 25368, 25381, 25385, 25407, 25435, 25448, 25461, 25486, 25497, 25507, 25526, 25539, 25552, 25555, 25567, 25590, 25619, 25633, 25650, 25670, 25681, 25687, 25697, 25739, 25746, 25777, 25792, 25800, 25817, 25833, 25837, 25846, 25883, 25892, 25901, 25906, 25910, 25921, 25925, 25940, 25943, 25950, 25961, 26045, 26091, 26109, 26112, 26126, 26139, 26189, 26207, 26278, 26290, 26319, 26321, 26325, 26327, 26385, 26395, 26405, 26417, 26600, 26617, 26627, 26793, 26794, 26795, 26984, 26987, 26995, 27005, 27013, 28046, 28603, 28625, 28782, 28960, 29236, 30007, 30025, 30042, 30079, 30096, 30138, 30157, 30170, 30202, 30217, 30228, 30324, 30371, 30411, 30447, 30650, 30652, 30655, 30660, 30672, 30718, 30829, 30831, 31017, 31185, 31285, 32991, 33030, 33302, 35510, 41954, 41962, 42000
 - `\exp_after:wN` ... 42–44, 214, 398, 402, 424, 427, 464, 482, 572, 593, 594, 641, 738, 751, 755, 898, 923, 936, 960, 1073, 1097, 1098, 1100, 1101, 1169, 1170, 1235, 1405, 1405, 1423, 1425, 1430, 1432, 1590, 1592, 1654, 1678, 1696, 1698, 1762, 1767, 1774, 1803, 1807, 1812, 1823, 1848, 1864, 1892, 1908, 1928, 1939, 1944, 2073, 2093, 2095, 2134, 2211, 2320, 2352, 2372, 2392, 2431, 2441, 2447, 2454, 2456, 2461, 2462, 2474, 2475, 2480, 2481, 2486, 2491, 2493, 2496, 2505, 2507, 2510, 2511, 2512, 2515, 2517, 2519, 2521, 2523, 2526, 2531, 2536, 2537, 2538, 2542, 2543, 2544, 2548, 2549, 2554, 2555, 2556, 2560, 2561, 2562, 2566, 2567, 2568, 2572, 2573, 2574, 2578, 2579, 2580, 2584, 2585, 2586, 2587, 2591, 2592, 2593, 2594, 2598, 2599, 2600, 2601, 2605, 2606, 2607, 2612, 2613, 2614, 2619, 2620, 2621, 2622, 2626, 2627, 2628, 2629, 2635, 2638, 2639, 2643, 2644, 2650, 2651, 2658, 2660, 2662, 2664, 2666, 2668, 2671, 2672, 2677, 2678, 2682, 2685, 2686, 2690, 2693, 2694, 2695, 2700, 2701, 2702, 2711, 2714, 2715, 2716, 2717, 2722, 2724, 2726, 2727, 2731, 2734, 2739, 2777, 2781, 2803, 2810, 2830, 2973, 2975, 3028, 3030, 3047, 3065, 3076, 3219, 3305, 3306, 3349, 3368, 3369, 3384, 3385, 3386, 3629, 3641, 3643, 3656, 3742, 3743, 3744, 3745, 3751, 3752, 3768, 3786, 3788, 3794, 3795, 3826, 3828, 3830, 3860, 3875, 3877, 3878, 3880, 3909, 3920, 3928, 3938, 3948, 3950, 3952, 3982, 4022, 4031, 4034, 4035, 4037, 4038, 4046, 4047, 4061, 4099, 4139, 4143, 4144, 4145, 4147, 4149, 4159, 4162, 4179, 4181, 4183, 4204, 4208, 4240, 4242, 4247, 4248, 4249, 4283, 4320, 4366, 4392, 4396, 4404,

4461, 4468, 4475, 4479, 4486, 4492,
4539, 4661, 4664, 4705, 4723, 4728,
4730, 4731, 4738, 4741, 4742, 4748,
4760, 4772, 4791, 4799, 4890, 4911,
4929, 4940, 4960, 5055, 5252, 5295,
5359, 5371, 5503, 5506, 5516, 5518,
5700, 5707, 5715, 5800, 5894, 6310,
6600, 6606, 6612, 6722, 6746, 6777,
6808, 6834, 6872, 6922, 6923, 6934,
7051, 7080, 7142, 7143, 7146, 7147,
7155, 7157, 7158, 7181, 7184, 7261,
7631, 7632, 7645, 7668, 7669, 7680,
7719, 7832, 7833, 7924, 7931, 7935,
7936, 7937, 7977, 8015, 8032, 8033,
8034, 8479, 8498, 8503, 8507, 8652,
8982, 8983, 8984, 9085, 9203, 9469,
9665, 9666, 10213, 10214, 10300,
10484, 10502, 10543, 10787, 10790,
10841, 10850, 10853, 10856, 10857,
10859, 10899, 10965, 10996, 11017,
11029, 11057, 11065, 11156, 11157,
11158, 11196, 11549, 11669, 12216,
12281, 12282, 12289, 12290, 12306,
12310, 12322, 12327, 12332, 12339,
12346, 12347, 12353, 12358, 12363,
12370, 12377, 12378, 12394, 12398,
12403, 12409, 12417, 12418, 12422,
12426, 12431, 12437, 12445, 12446,
12472, 12496, 12497, 12498, 12499,
12500, 12559, 12560, 12561, 12603,
12645, 12646, 12707, 12717, 12722,
12798, 12829, 12843, 12891, 12893,
13005, 13058, 13060, 13062, 13064,
13156, 13164, 13167, 13214, 13224,
13248, 13258, 13259, 13260, 13263,
13267, 13268, 13292, 13359, 13440,
13442, 13443, 13444, 13449, 13450,
13452, 13524, 13542, 13543, 13602,
13608, 13623, 13626, 13641, 13643,
13644, 13661, 13669, 13674, 13676,
13679, 13730, 13735, 13740, 13745,
13931, 13932, 13944, 14009, 14032,
14067, 14068, 14079, 14080, 14088,
14096, 14098, 14105, 14110, 14128,
14129, 14130, 14142, 14143, 14170,
14172, 14178, 14184, 14198, 14218,
14230, 14247, 14255, 14263, 14270,
14277, 14289, 14491, 14507, 14526,
14535, 14556, 14557, 14562, 14563,
14588, 14589, 14604, 14605, 14653,
14658, 14725, 15054, 15090, 15092,
15107, 15113, 15278, 15280, 15347,
15366, 15367, 15381, 15382, 15409,
15410, 15525, 15547, 15560, 15561,
15588, 15683, 15704, 15713, 16641,
16648, 16816, 17071, 17077, 17079,
17103, 17154, 17155, 17234, 17268,
17323, 17331, 17342, 17536, 17538,
17550, 17558, 17666, 17676, 17762,
17796, 17808, 17821, 17822, 17823,
17845, 17846, 17891, 17926, 17927,
17928, 18029, 18030, 18032, 18033,
18041, 18042, 18046, 18049, 18092,
18093, 18119, 18120, 18123, 18176,
18217, 18231, 18236, 18239, 18240,
18247, 18248, 18264, 18265, 18286,
18287, 18296, 18433, 18438, 18443,
18466, 18468, 18628, 18629, 18630,
18639, 18658, 18659, 18847, 18875,
18880, 18908, 18921, 18931, 18958,
18960, 18961, 18969, 18986, 19030,
19107, 19140, 19142, 19148, 19151,
19196, 19204, 19266, 19344, 19359,
19381, 19395, 19450, 19458, 19464,
19545, 19567, 19577, 19695, 19696,
19699, 19700, 19992, 19993, 20029,
20072, 20073, 20075, 20076, 20077,
20239, 20258, 20306, 20443, 20472,
20473, 20475, 20481, 20484, 20566,
20569, 20585, 20591, 20594, 20597,
20604, 20605, 20607, 20643, 20645,
20655, 20656, 20657, 20659, 20665,
20666, 20668, 20675, 20676, 20678,
20705, 20710, 20712, 20718, 20761,
20766, 20773, 20797, 20842, 20853,
20861, 20862, 20892, 20917, 20921,
20924, 20941, 20961, 21037, 21038,
21045, 21053, 21087, 21093, 21094,
21096, 21110, 21111, 21114, 21132,
21140, 21144, 21151, 21153, 21162,
21168, 21287, 21292, 21299, 21301,
21309, 21339, 21358, 21360, 21361,
21366, 21458, 21481, 21486, 21493,
21560, 21570, 21646, 21650, 21653,
21654, 21661, 21662, 21710, 21715,
21726, 21729, 21836, 21837, 21838,
21893, 21913, 21933, 21956, 21964,
21965, 21986, 21991, 21999, 22005,
22020, 22116, 22555, 22570, 22610,
22755, 22790, 22804, 22908, 22909,
22910, 23278, 23489, 23490, 23569,
23592, 24059, 24060, 24061, 24079,
24087, 24111, 24112, 24154, 24161,
24162, 24173, 24269, 24271, 24272,
24290, 24291, 24292, 24302, 24304,
24312, 24314, 24321, 24322, 24323,
24324, 24325, 24326, 24331, 24332,
24333, 24334, 24335, 24336, 24337,

24380, 24393, 24396, 24407, 24409,
24424, 24428, 24429, 24430, 24433,
24434, 24499, 24501, 24528, 24532,
24558, 24562, 24579, 24586, 24588,
24660, 24668, 24685, 24698, 24744,
24819, 24888, 24889, 24890, 24950,
24960, 24981, 24988, 25007, 25009,
25011, 25022, 25023, 25026, 25037,
25041, 25048, 25049, 25060, 25061,
25070, 25077, 25079, 25080, 25088,
25117, 25135, 25136, 25139, 25140,
25142, 25143, 25147, 25148, 25150,
25151, 25160, 25161, 25166, 25172,
25178, 25186, 25194, 25213, 25214,
25217, 25218, 25220, 25227, 25228,
25230, 25248, 25249, 25277, 25280,
25285, 25286, 25291, 25292, 25294,
25303, 25304, 25305, 25306, 25309,
25310, 25311, 25314, 25329, 25346,
25347, 25357, 25358, 25368, 25380,
25384, 25397, 25399, 25407, 25417,
25419, 25425, 25430, 25432, 25434,
25440, 25441, 25445, 25447, 25459,
25460, 25483, 25485, 25491, 25494,
25496, 25500, 25505, 25510, 25511,
25521, 25522, 25524, 25525, 25528,
25532, 25537, 25551, 25554, 25566,
25575, 25582, 25583, 25584, 25585,
25587, 25589, 25600, 25601, 25602,
25603, 25605, 25607, 25609, 25610,
25611, 25617, 25618, 25628, 25632,
25633, 25635, 25636, 25637, 25642,
25648, 25649, 25660, 25662, 25669,
25670, 25672, 25673, 25680, 25686,
25696, 25760, 25773, 25774, 25775,
25776, 25790, 25791, 25793, 25798,
25799, 25814, 25816, 25833, 25837,
25846, 25880, 25881, 25882, 25888,
25889, 25890, 25891, 25897, 25898,
25899, 25900, 25908, 25909, 25923,
25924, 25932, 25938, 25939, 25941,
25942, 25948, 25949, 25951, 25977,
25990, 26010, 26011, 26013, 26014,
26021, 26022, 26024, 26027, 26039,
26040, 26041, 26043, 26044, 26045,
26052, 26053, 26055, 26056, 26063,
26064, 26066, 26069, 26084, 26085,
26086, 26089, 26090, 26091, 26101,
26102, 26103, 26106, 26107, 26108,
26111, 26115, 26124, 26125, 26136,
26137, 26138, 26141, 26143, 26144,
26145, 26172, 26173, 26175, 26176,
26177, 26187, 26188, 26189, 26191,
26192, 26193, 26205, 26206, 26209,
26211, 26212, 26213, 26233, 26234,
26235, 26236, 26237, 26238, 26239,
26249, 26250, 26252, 26253, 26254,
26260, 26271, 26272, 26273, 26274,
26275, 26276, 26277, 26278, 26283,
26284, 26285, 26286, 26287, 26288,
26289, 26305, 26306, 26308, 26309,
26316, 26317, 26318, 26323, 26324,
26326, 26342, 26357, 26366, 26376,
26382, 26383, 26384, 26389, 26402,
26403, 26404, 26410, 26599, 26616,
26626, 26662, 26663, 26710, 26792,
26896, 26898, 26938, 26940, 26943,
26979, 26981, 26983, 26986, 26993,
26994, 26997, 26998, 27003, 27004,
27011, 27012, 27047, 27048, 27049,
27051, 27062, 27087, 27089, 27095,
27096, 27100, 27103, 27125, 27127,
27140, 27142, 27148, 27150, 27153,
27159, 27161, 27163, 27164, 27165,
27167, 27174, 27176, 27178, 27183,
27186, 27192, 27193, 27197, 27199,
27200, 27201, 27209, 27211, 27212,
27219, 27225, 27232, 27233, 27238,
27239, 27240, 27241, 27261, 27262,
27263, 27269, 27270, 27271, 27276,
27278, 27286, 27288, 27290, 27291,
27293, 27305, 27307, 27309, 27310,
27315, 27367, 27368, 27375, 27376,
27378, 27380, 27382, 27385, 27388,
27390, 27392, 27401, 27403, 27410,
27412, 27414, 27415, 27416, 27423,
27425, 27427, 27428, 27429, 27451,
27452, 27455, 27463, 27465, 27469,
27470, 27471, 27472, 27477, 27479,
27486, 27489, 27492, 27495, 27504,
27507, 27510, 27513, 27520, 27522,
27529, 27538, 27540, 27542, 27559,
27561, 27568, 27570, 27573, 27579,
27581, 27583, 27584, 27585, 27587,
27601, 27602, 27605, 27623, 27625,
27627, 27639, 27642, 27645, 27648,
27651, 27654, 27657, 27660, 27664,
27676, 27680, 27684, 27687, 27702,
27708, 27710, 27712, 27722, 27746,
27749, 27761, 27763, 27767, 27768,
27769, 27771, 27772, 27774, 27781,
27789, 27790, 27796, 27797, 27803,
27806, 27807, 27808, 27809, 27817,
27866, 27871, 27873, 27881, 27884,
27887, 27890, 27893, 27896, 27904,
27905, 27917, 27925, 27927, 27937,
27939, 27946, 27956, 27958, 27961,
27964, 27967, 27970, 27983, 27985,

27994, 27996, 28004, 28006, 28016,
 28019, 28022, 28029, 28044, 28045,
 28063, 28065, 28066, 28125, 28138,
 28140, 28147, 28160, 28162, 28164,
 28189, 28203, 28205, 28212, 28214,
 28257, 28258, 28259, 28261, 28262,
 28263, 28265, 28266, 28272, 28274,
 28275, 28281, 28283, 28284, 28285,
 28286, 28298, 28304, 28306, 28352,
 28359, 28366, 28386, 28387, 28389,
 28391, 28393, 28406, 28411, 28412,
 28413, 28414, 28415, 28419, 28425,
 28427, 28434, 28440, 28442, 28443,
 28450, 28451, 28452, 28453, 28454,
 28455, 28456, 28457, 28464, 28466,
 28468, 28470, 28472, 28477, 28479,
 28481, 28483, 28485, 28487, 28509,
 28513, 28522, 28523, 28528, 28530,
 28539, 28542, 28543, 28544, 28546,
 28547, 28548, 28556, 28562, 28574,
 28577, 28578, 28580, 28581, 28605,
 28606, 28609, 28611, 28627, 28631,
 28632, 28633, 28649, 28655, 28721,
 28722, 28723, 28730, 28732, 28733,
 28738, 28740, 28741, 28750, 28751,
 28753, 28756, 28759, 28777, 28781,
 28782, 28786, 28788, 28823, 28829,
 28830, 28832, 28834, 28835, 28837,
 28838, 28848, 28849, 28852, 28853,
 28854, 28856, 28857, 28858, 28876,
 28877, 28879, 28880, 28886, 28888,
 28891, 28894, 28897, 28900, 28908,
 28911, 28913, 28916, 28923, 28927,
 28935, 28936, 28939, 28941, 28943,
 28948, 28949, 28956, 28961, 28962,
 28970, 28971, 28972, 28973, 29016,
 29038, 29039, 29042, 29043, 29052,
 29053, 29054, 29058, 29065, 29066,
 29067, 29208, 29209, 29210, 29212,
 29230, 29231, 29232, 29233, 29234,
 29235, 29242, 29251, 29258, 29259,
 29483, 29484, 29490, 29491, 29494,
 29499, 29502, 29505, 29508, 29511,
 29514, 29517, 29520, 29536, 29537,
 29547, 29556, 29564, 29565, 29567,
 29568, 29573, 29574, 29583, 29590,
 29599, 29600, 29613, 29615, 29660,
 29661, 29670, 29673, 29708, 29714,
 29715, 29757, 29758, 29760, 29774,
 29775, 29783, 29794, 29828, 29831,
 29842, 29843, 29846, 29848, 29854,
 29868, 29870, 29911, 29914, 29934,
 30007, 30020, 30024, 30042, 30045,
 30066, 30067, 30074, 30078, 30096,
 30099, 30128, 30129, 30135, 30136,
 30137, 30144, 30152, 30156, 30170,
 30173, 30189, 30190, 30197, 30201,
 30212, 30216, 30228, 30233, 30234,
 30235, 30241, 30243, 30246, 30248,
 30313, 30323, 30330, 30335, 30336,
 30346, 30373, 30384, 30386, 30388,
 30390, 30395, 30396, 30398, 30409,
 30410, 30430, 30436, 30437, 30439,
 30442, 30447, 30453, 30454, 30484,
 30486, 30489, 30492, 30494, 30502,
 30504, 30508, 30512, 30519, 30524,
 30536, 30570, 30590, 30608, 30649,
 30653, 30657, 30659, 30670, 30671,
 30677, 30697, 30717, 30817, 30826,
 30827, 30828, 30832, 30833, 31011,
 31012, 31013, 31014, 31015, 31016,
 31019, 31021, 31061, 31062, 31063,
 31067, 31068, 31081, 31092, 31095,
 31099, 31104, 31108, 31155, 31156,
 31180, 31181, 31182, 31183, 31184,
 31192, 31203, 31209, 31235, 31236,
 31237, 31244, 31245, 31252, 31253,
 31261, 31262, 31266, 31267, 31268,
 31273, 31278, 31284, 31287, 31288,
 31289, 31290, 31291, 31426, 31717,
 31718, 31928, 31967, 31981, 31998,
 32327, 32456, 32458, 32695, 32697,
 32704, 32705, 32708, 32737, 32739,
 32745, 32757, 32765, 32767, 32884,
 32889, 32890, 32893, 32894, 32899,
 32906, 32909, 32912, 32989, 33028,
 33045, 33046, 33090, 33091, 33114,
 33167, 33187, 33266, 33292, 33301,
 33316, 33318, 34923, 35508, 35557,
 35558, 35619, 35620, 35621, 35628,
 35703, 38634, 38638, 38657, 38658,
 39169, 39232, 39324, 39351, 39356,
 39361, 39366, 39512, 39530, 39693,
 39970, 40574, 41856, 41891, 41892,
 41904, 41962, 42000, 42062, 42732
 \exp_args:cc 38, 1422, 1424, 2504
 \exp_args:Nc 35, 38, 417, 1422, 1422,
 1426, 1434, 1702, 1715, 1723, 1731,
 1982, 2005, 2043, 2048, 2055, 2066,
 2113, 2125, 2210, 2255, 2256, 2257,
 2258, 2280, 2284, 2504, 2774, 3976,
 5656, 10218, 10224, 10470, 12686,
 12958, 13589, 13651, 13656, 13664,
 13697, 16805, 16885, 16943, 19171,
 23678, 24969, 25210, 26095, 26119,
 26149, 26151, 26153, 26155, 26157,
 26159, 26161, 26163, 26181, 26197,
 26198, 26217, 26219, 30694, 31626,

- 31627, 31628, 31656, 32401, 35729,
 37712, 37743, 39371, 41179, 41860
 \exp_args:Ncc 39, 2045,
 2049, 2057, 2263, 2264, 2265, 2266,
 2504, 2506, 14673, 14855, 17243, 17279
 \exp_args:Nccc 39, 2504, 2508
 \exp_args:Ncco 39, 2589, 2624
 \exp_args:Nccx 41, 3148
 \exp_args:Nce 39, 42656, 42680
 \exp_args:Ncee 40
 \exp_args:NceV 40
 \exp_args:Ncev 40
 \exp_args:Ncf 39, 2534, 2576
 \exp_args:NcNc 39, 2589, 2610
 \exp_args:Ncnc 40
 \exp_args:Ncne 40
 \exp_args:NcNo 39, 2589, 2617
 \exp_args:Ncno 40, 3148
 \exp_args:NcnV 40, 3148
 \exp_args:Ncnv 40
 \exp_args:Ncnx 41, 3148
 \exp_args:Nco 39, 429, 2534, 2558
 \exp_args:Ncoo 40, 3148
 \exp_args:NcV 39, 2534, 2564
 \exp_args:Ncv 39, 2534, 2570
 \exp_args:NcVe 40
 \exp_args:Ncve 40
 \exp_args:NcVV 40, 3148
 \exp_args:NcVv 40
 \exp_args:NcvV 40
 \exp_args:Ncvv 40
 \exp_args:Ncx 39, 3122
 \exp_args:Ne 38, 2071, 2520,
 2520, 4565, 5346, 5347, 5718, 6027,
 6029, 6449, 8449, 8450, 9442, 10375,
 10378, 10619, 10622, 10692, 11049,
 11051, 11175, 11189, 11218, 11308,
 11317, 11338, 11348, 11358, 11371,
 11559, 11580, 12617, 13523, 14390,
 14690, 15747, 15760, 19120, 19173,
 20060, 22965, 23001, 23031, 23063,
 23676, 26498, 31045, 31681, 32099,
 32113, 32150, 32574, 32651, 33128,
 33234, 33244, 33381, 33391, 33422,
 33581, 33787, 33830, 33906, 33922,
 33976, 34161, 34183, 34511, 34636,
 34651, 34718, 35178, 35338, 35382,
 35448, 35511, 35663, 36084, 36088,
 37620, 38721, 38752, 38954, 38961,
 39091, 39426, 39846, 39949, 40069,
 40215, 40220, 40619, 40857, 40900,
 40907, 40912, 40980, 40987, 40992,
 41064, 41070, 42026, 42050, 42500
 \exp_args:Nee ... 39, 3122, 10222,
 11445, 11611, 38967, 39618, 41479
 \exp_args:Neee
 40, 3148, 11323, 38714,
 38994, 39077, 39136, 39143, 39150
 \exp_args:Nf 38, 2101,
 2522, 2522, 8474, 9495, 9496, 9824,
 9826, 10891, 10949, 12540, 12579,
 12585, 13384, 13385, 13401, 13421,
 13429, 13433, 13457, 13463, 13473,
 13520, 13555, 14057, 14059, 14118,
 14120, 14136, 14570, 14575, 14910,
 14938, 15463, 15464, 15628, 15639,
 16805, 16807, 16831, 16837, 16885,
 16892, 16943, 16953, 16987, 17012,
 17894, 17895, 17911, 18527, 18532,
 18537, 18542, 18720, 18790, 18792,
 18810, 18819, 18830, 18839, 18977,
 18994, 19770, 19784, 19806, 19817,
 19873, 19943, 19949, 19955, 19961,
 20787, 20789, 20899, 21528, 21751,
 21756, 21761, 21766, 24130, 26849,
 30003, 30549, 30585, 30733, 31433,
 31562, 31735, 31790, 32004, 32335,
 32343, 32448, 32466, 34243, 34272,
 34306, 34384, 34402, 34419, 34444,
 38896, 39933, 41319, 41945, 41964
 \exp_args:Nff
 39, 3122, 13427, 17636, 24599
 \exp_args:Nffo 40, 3148
 \exp_args:Nfo 39, 3122
 \exp_args:NNc . 39, 393, 2044, 2047,
 2056, 2127, 2259, 2260, 2261, 2262,
 2297, 2300, 2504, 2504, 3030, 10300,
 10453, 10454, 10543, 17284, 18022,
 18676, 18687, 21867, 21874, 26859,
 26866, 30846, 30981, 31132, 31658
 \exp_args:Nnc . 39, 3122, 35487, 35497
 \exp_args:NNcc 40
 \exp_args:NNcf 40, 3148
 \exp_args:NNe 39, 2305, 2534, 2546,
 5824, 11638, 11652, 11718, 22442,
 32366, 39761, 42512, 42573, 42621
 \exp_args:Nne ... 39, 3122, 11182,
 11233, 11268, 13311, 41764, 41774
 \exp_args:NNee 40
 \exp_args:NNee 40, 39159
 \exp_args:NNeV 40
 \exp_args:NNev 40
 \exp_args:NNf . 39, 964, 2534, 2552,
 6720, 10299, 10542, 10774, 20931,
 21860, 29202, 29203, 41344, 41852
 \exp_args:Nnf
 39, 3122, 12518, 23674, 32437

- \exp_args:Nnff [40](#), [3148](#)
- \exp_args:Nnnc [40](#), [3148](#)
- \exp_args:NNNe
..... [39](#), [447](#), [2589](#), [2603](#), [5018](#), [5499](#)
- \exp_args:NNne [40](#)
- \exp_args:Nnnf [40](#), [11565](#)
- \exp_args:Nnnf [40](#), [3148](#)
- \exp_args:NNnne [35281](#)
- \exp_args:Nnnne [35097](#)
- \exp_args:NNNo
..... [39](#), [483](#), [2515](#), [2518](#), [4218](#),
[5010](#), [6121](#), [6184](#), [7535](#), [12604](#), [12626](#)
- \exp_args:NNno [40](#), [3148](#)
- \exp_args:Nnno [40](#), [3148](#), [32339](#)
- \exp_args:NNNV [39](#), [2589](#), [2589](#), [32975](#),
[38808](#), [39411](#), [39503](#), [40727](#), [40761](#)
- \exp_args:NNNv . [39](#), [2589](#), [2596](#), [10456](#)
- \exp_args:NNnV [40](#), [3148](#)
- \exp_args:NNnv [40](#)
- \exp_args:NnNV [40](#)
- \exp_args:NnnV [40](#)
- \exp_args:Nnnv [40](#), [32340](#), [32414](#)
- \exp_args:NNNx [41](#), [3148](#)
- \exp_args:NNnx [41](#), [3148](#)
- \exp_args:Nnnx [41](#), [3148](#)
- \exp_args:NNo
. [33](#), [39](#), [2515](#), [2516](#), [7001](#), [12476](#),
[20949](#), [21108](#), [23487](#), [31027](#), [41301](#)
- \exp_args:Nno [39](#), [3122](#), [7021](#), [8990](#),
[10427](#), [11165](#), [12503](#), [12564](#), [12800](#),
[12809](#), [13162](#), [13255](#), [13289](#), [13999](#),
[21718](#), [24643](#), [24651](#), [24662](#), [24679](#),
[24687](#), [24719](#), [25236](#), [25240](#), [41209](#)
- \exp_args:NNoo [40](#), [3148](#)
- \exp_args:NNox [41](#), [3148](#)
- \exp_args:Nnox [41](#), [3148](#)
- \exp_args:NNV [39](#), [2534](#), [2534](#)
- \exp_args:NNv [39](#), [2534](#), [2540](#)
- \exp_args:NnV
..... [39](#), [3122](#), [38875](#), [40640](#), [40681](#)
- \exp_args:Nnv [39](#), [3122](#)
- \exp_args:NNVe [40](#)
- \exp_args:NNve [40](#)
- \exp_args:NNVV [40](#), [3148](#)
- \exp_args:NNVv [40](#)
- \exp_args:NNvV [40](#)
- \exp_args:NNvv [40](#)
- \exp_args:NNx [39](#), [3122](#)
- \exp_args:Nnx [39](#), [3122](#)
- \exp_args:No
..... [35](#), [38](#), [116](#), [755](#), [2289](#), [2294](#),
[2515](#), [2515](#), [2996](#), [3019](#), [3026](#), [3108](#),
[3125](#), [3971](#), [4005](#), [4065](#), [4067](#), [4400](#),
[5100](#), [5164](#), [5184](#), [5869](#), [5918](#), [6423](#),
[6845](#), [6850](#), [7043](#), [7058](#), [7153](#), [7351](#),
[7749](#), [7753](#), [7778](#), [7779](#), [7974](#), [8965](#),
[8980](#), [9659](#), [10486](#), [10686](#), [10782](#),
[11153](#), [11253](#), [12491](#), [12854](#), [12855](#),
[12856](#), [12883](#), [12884](#), [12885](#), [12886](#),
[12887](#), [12937](#), [12967](#), [12977](#), [12998](#),
[13069](#), [13071](#), [13073](#), [13078](#), [13080](#),
[13082](#), [13084](#), [13086](#), [13088](#), [13161](#),
[13170](#), [13377](#), [13379](#), [13403](#), [13410](#),
[13414](#), [13471](#), [13480](#), [13938](#), [13965](#),
[13970](#), [13984](#), [14005](#), [14053](#), [14064](#),
[14114](#), [14125](#), [14192](#), [14211](#), [14251](#),
[14266](#), [14783](#), [14836](#), [15122](#), [16759](#),
[17475](#), [18796](#), [18802](#), [19457](#), [19469](#),
[19471](#), [19506](#), [19511](#), [19686](#), [19800](#),
[19804](#), [19838](#), [22122](#), [22967](#), [23003](#),
[23033](#), [23065](#), [23175](#), [23487](#), [23520](#),
[26497](#), [30867](#), [30882](#), [30902](#), [30963](#),
[31042](#), [31111](#), [31424](#), [32732](#), [36510](#),
[41216](#), [41689](#), [41697](#), [42504](#), [42740](#)
- \exp_args:Noc [39](#), [3122](#)
- \exp_args:Nof [39](#), [3122](#), [12533](#)
- \exp_args:Noo ... [39](#), [3122](#), [5206](#), [6436](#)
- \exp_args:Noof [40](#), [3148](#)
- \exp_args:Nooo [40](#), [3148](#)
- \exp_args:Noooo [10224](#)
- \exp_args:Noox [41](#), [3148](#)
- \exp_args:Nox [39](#), [3122](#)
- \exp_args:NV [38](#), [2522](#), [2529](#),
[10925](#), [11001](#), [11140](#), [11698](#), [11703](#),
[12610](#), [22795](#), [22798](#), [22963](#), [22999](#),
[23029](#), [23061](#), [31671](#), [32974](#), [33506](#),
[34975](#), [38766](#), [39376](#), [39665](#), [39678](#),
[39776](#), [39800](#), [39842](#), [40187](#), [40629](#),
[40639](#), [40644](#), [40752](#), [41925](#), [41927](#)
- \exp_args:Nv [38](#),
[2522](#), [2524](#), [32427](#), [33280](#), [38765](#), [40258](#)
- \exp_args:NNNV [40](#)
- \exp_args:NVo [39](#), [3122](#)
- \exp_args:NVV .. [39](#), [2534](#), [2582](#), [10836](#)
- \exp_args:Nx [38](#), [2631](#),
[2631](#), [3039](#), [22969](#), [23005](#), [23035](#), [23067](#)
- \exp_args:Nxo [39](#), [3122](#)
- \exp_args:Nxx [39](#), [3122](#)
- \exp_args_generate:n
. [36](#), [3106](#), [3106](#), [10219](#), [11571](#), [35278](#)
- \exp_arsgs:Nn [40128](#)
- \exp_end: [44](#), [398](#), [401](#), [405](#), [426](#), [427](#),
[434](#), [592–594](#), [615](#), [737](#), [745](#), [746](#),
[754](#), [755](#), [769](#), [1098](#), [1128](#), [1409](#),
[1703](#), [1716](#), [1724](#), [1732](#), [2493](#), [2502](#),
[2742](#), [3030](#), [7931](#), [7936](#), [7985](#), [8015](#),
[8026](#), [8033](#), [8647](#), [8683](#), [8686](#), [8687](#),
[8688](#), [8689](#), [8690](#), [8691](#), [8692](#), [8693](#),

- 8694, 8696, 12567, 13331, 13452,
13595, 13614, 13928, 14113, 18553,
19192, 20019, 20026, 20029, 20068,
20072, 20534, 21777, 24950, 25915,
28627, 30373, 30375, 30447, 30653,
30670, 30832, 33010, 35530, 41959
`\exp_end_continue_f:nw` 45, 2742, 2750
`\exp_end_continue_f:w` 44,
45, 426, 1100, 1101, 2462, 2523,
2556, 2580, 2651, 2666, 2679, 2703,
2718, 2731, 2742, 2744, 8481, 10142,
10858, 12907, 19266, 20586, 21715,
24316, 24431, 24435, 25070, 25088,
25109, 25173, 25178, 25186, 25194,
25215, 25293, 25329, 25337, 25368,
25739, 25746, 25792, 25839, 25846,
25883, 25931, 25937, 25940, 25950,
25961, 26126, 26319, 26321, 26325,
26327, 26385, 26395, 26405, 26417,
26600, 26617, 26627, 26793, 26794,
26795, 26984, 26995, 27005, 27013,
28046, 28782, 28960, 29236, 30007,
30025, 30042, 30079, 30096, 30138,
30157, 30170, 30202, 30217, 30228,
30324, 30411, 30655, 30660, 30718,
31017, 31185, 31285, 33302, 42000
`\exp_last_two_unbraced:Nnn`
..... 41, 2721, 2721, 21541
`\exp_last_unbraced:cf`
..... 38678, 38684, 38690
`\exp_last_unbraced:Nco`
..... 41, 2658, 2681, 19588
`\exp_last_unbraced:NcV` 41, 2658, 2683
`\exp_last_unbraced:Ne`
..... 41, 2658, 2663, 22442, 30281
`\exp_last_unbraced:Nf`
..... 41, 2658, 2665,
4716, 6047, 15006, 15292, 15481,
15653, 16654, 16765, 16842, 16897,
16965, 16999, 17029, 17172, 17193,
17314, 17336, 18808, 18828, 20988,
21406, 24144, 24159, 24594, 26591,
27077, 30603, 30815, 38668, 41923
`\exp_last_unbraced:Nfo`
..... 41, 2658, 2708, 31135
`\exp_last_unbraced:NNf` 41, 2658, 2675
`\exp_last_unbraced:Nnf`
..... 41, 2658, 2706, 21379, 21417
`\exp_last_unbraced:NNnf`
..... 41, 2658, 2698, 8394
`\exp_last_unbraced:NNNf`
..... 41, 2658, 2712, 8399
`\exp_last_unbraced:NNNNo` 41,
2658, 2710, 2790, 2794, 2957, 11067,
14309, 15123, 24384, 24402, 32773
`\exp_last_unbraced:NNNo`
..... 41, 2658, 2689, 21537, 21575
`\exp_last_unbraced:NnNo` 41, 2658, 2709
`\exp_last_unbraced:NNNV` 41, 2658, 2691
`\exp_last_unbraced:NNo`
..... 41, 2658, 2667,
10689, 13339, 33036, 35545, 38282
`\exp_last_unbraced:Nno`
..... 41, 2658, 2705, 17979, 19617
`\exp_last_unbraced:NNV` 41, 2658, 2669
`\exp_last_unbraced:No` 41,
2658, 2658, 38444, 38449, 38518, 38524
`\exp_last_unbraced:Noo` 41, 2658, 2707
`\exp_last_unbraced:NV`
..... 41, 2658, 2659, 8032, 40200
`\exp_last_unbraced:Nv`
..... 41, 1286, 2658, 2661
`\exp_last_unbraced:Nx` 41, 2658, 2720
`\exp_not:N` .. 42, 101, 172, 287, 427,
433, 464, 474, 482, 563, 591, 741–
743, 935, 936, 942, 953, 1106, 1405,
1406, 1658, 1749, 1752, 2133, 2134,
2210, 2211, 2320, 2447, 2486, 2632,
2726, 2726, 2767, 2769, 2770, 2777,
2778, 2779, 2780, 2781, 2782, 2788,
2814, 2823, 2878, 2879, 2939, 2945,
2947, 2953, 3000, 3017, 3019, 3047,
3641, 3642, 3643, 3644, 3645, 3646,
3730, 3733, 3886, 3887, 3888, 3889,
3890, 3891, 3892, 3893, 3894, 3895,
3896, 3897, 3898, 3899, 3900, 3901,
3963, 3964, 4127, 4136, 4137, 4139,
4145, 4156, 4160, 4169, 4170, 4171,
4173, 4195, 4199, 4283, 4285, 4287,
4293, 4295, 4339, 4692, 4694, 4696,
4698, 4700, 4702, 5088, 5090, 5288,
5290, 5301, 5305, 5463, 6136, 6839,
7253, 7266, 8009, 8426, 8432, 9135,
9136, 9540, 9542, 9544, 9546, 9672,
9674, 9678, 9680, 9685, 9687, 10324,
10325, 10329, 11150, 11153, 11154,
11156, 11157, 11158, 11159, 11162,
11163, 11260, 11262, 11520, 11521,
11522, 11523, 11524, 11527, 11528,
12626, 12627, 12629, 12710, 12749,
12750, 12751, 12752, 13177, 13195,
13223, 13230, 13241, 13242, 13314,
13317, 13318, 13956, 13957, 14330,
14333, 14335, 14336, 14337, 14340,
14900, 14901, 14928, 14929, 14930,
15767, 15768, 15769, 15771, 15772,
15774, 16659, 16660, 16662, 16674,
16677, 16678, 17474, 17476, 17520,

17521, 17522, 17523, 18003, 18074,
 18692, 18957, 19520, 19979, 20034,
 20036, 20038, 20039, 20040, 20042,
 20044, 20046, 20047, 20049, 20051,
 20053, 20055, 20063, 20077, 20097,
 20100, 20103, 20106, 20109, 20112,
 20115, 20118, 20121, 20124, 20161,
 20165, 20170, 20175, 20180, 20187,
 20194, 20199, 20204, 20209, 20214,
 20219, 20226, 20231, 20236, 20239,
 20240, 20243, 20253, 20258, 20273,
 20292, 20297, 20298, 20299, 20300,
 20301, 20302, 20304, 20306, 20307,
 20308, 20312, 20313, 20316, 20317,
 20338, 20341, 20353, 20357, 20437,
 20440, 20441, 20443, 20444, 20448,
 20451, 20452, 20454, 20457, 20577,
 20590, 20604, 20617, 20623, 20654,
 20657, 20664, 20665, 20674, 20675,
 20689, 20690, 20701, 20702, 20821,
 20823, 20827, 20828, 20914, 21163,
 21275, 21310, 21316, 21327, 21462,
 21500, 21505, 21506, 21522, 21525,
 21527, 21528, 21529, 21532, 21694,
 21695, 21699, 21700, 21704, 21705,
 21879, 21918, 22610, 22611, 22613,
 22617, 22618, 22640, 22642, 22696,
 22697, 22698, 22714, 22779, 22781,
 22808, 22809, 22930, 22931, 23164,
 23166, 23172, 23318, 23323, 23692,
 23697, 23701, 23704, 23713, 23714,
 24380, 24381, 25134, 25135, 25230,
 25231, 25232, 25233, 25339, 25379,
 25383, 25405, 25499, 25531, 25616,
 25630, 25647, 25658, 25668, 25705,
 25707, 25810, 25811, 25813, 25814,
 25815, 25816, 25817, 25818, 25821,
 25823, 25825, 26008, 26009, 26050,
 26051, 26171, 26186, 26871, 27030,
 29089, 29090, 29091, 29095, 29096,
 29097, 30570, 30571, 30572, 30574,
 30579, 30709, 30710, 30988, 30989,
 30990, 31052, 31053, 31054, 31080,
 31722, 31941, 31973, 32689, 32690,
 32693, 32704, 32778, 32781, 32784,
 32787, 32790, 32793, 32796, 32830,
 32833, 32835, 32836, 32837, 32840,
 32881, 32884, 32885, 32888, 32889,
 32890, 32891, 32892, 32893, 32894,
 32895, 32896, 32898, 32899, 32900,
 32937, 33079, 33207, 33208, 33213,
 33214, 33244, 33316, 33473, 33556,
 33618, 34962, 35027, 37505, 37506,
 38747, 38748, 38749, 38750, 38751,
 38752, 38753, 38975, 39186, 39187,
 39188, 39189, 39190, 39191, 39211,
 39212, 39213, 39432, 39433, 39437,
 39439, 39544, 39545, 39646, 39652,
 39654, 39655, 39761, 39762, 39763,
 39764, 39919, 39920, 39921, 40058,
 40060, 40063, 40098, 40151, 40541,
 40544, 40546, 40551, 40555, 40558,
 40561, 40766, 40768, 40771, 40773,
 40776, 40779, 40783, 40786, 40791,
 41022, 41035, 41043, 41393, 41401,
 41417, 41420, 41649, 41652, 41655,
 41658, 41661, 41664, 42029, 42033,
 42043, 42050, 42053, 42084, 42087,
 42502, 42516, 42518, 42576, 42578,
 42624, 42626, 42629, 42631, 42659,
 42661, 42665, 42667, 42683, 42685
 \exp_not:n 42,
 43, 54, 101, 124–127, 160, 161, 166,
 167, 172, 196, 197, 199, 214, 224,
 259, 260, 301, 303, 383, 458, 463,
 464, 470, 474, 482, 483, 491, 563,
 570, 574, 585, 591, 716, 727, 749,
 755–757, 866, 869, 911, 912, 915,
 918, 928, 961, 1012, 1351, 1352,
 1355, 1553, 1405, 1407, 1659, 1665,
 1667, 1673, 1674, 1754, 2073, 2333,
 2334, 2447, 2458, 2469, 2647, 2655,
 2726, 2727, 2728, 2730, 2732, 2737,
 2883, 2898, 2913, 2988, 3021, 3044,
 3459, 3758, 3872, 3903, 4159, 4188,
 4200, 4202, 4208, 4221, 4339, 4419,
 5088, 5090, 5721, 6193, 6357, 6575,
 6763, 6828, 6840, 6918, 6919, 7181,
 7184, 7253, 7261, 7278, 7293, 7334,
 7526, 7654, 7662, 7690, 7695, 7697,
 7977, 9022, 9054, 9534, 9709, 10641,
 10660, 11838, 11841, 11843, 12327,
 12332, 12358, 12363, 12398, 12403,
 12426, 12431, 12711, 12712, 12713,
 13313, 13316, 13320, 13400, 13595,
 13596, 13672, 13775, 13820, 13831,
 13977, 17446, 17448, 17549, 17550,
 17557, 17558, 17574, 17606, 17675,
 17679, 17682, 17683, 17694, 17697,
 17700, 17801, 17833, 17857, 17910,
 18004, 18084, 18135, 18145, 18693,
 19232, 19275, 19276, 19290, 19292,
 19368, 19421, 19457, 19465, 19485,
 19521, 19712, 19717, 19745, 19748,
 19751, 19783, 19815, 19835, 20064,
 20311, 20558, 20578, 20618, 20624,
 20772, 20848, 20849, 20915, 20924,
 20942, 21114, 21119, 21126, 21167,

- 21168, 21173, 21276, 21282, 21283,
 21286, 21314, 21318, 21325, 21500,
 21508, 21558, 21569, 21880, 22362,
 22371, 22440, 22441, 22512, 22700,
 22714, 22729, 22779, 22781, 22811,
 22932, 23136, 23170, 23172, 23501,
 23511, 23537, 23539, 25225, 26458,
 26460, 26462, 26563, 26872, 27031,
 30292, 31073, 31074, 31078, 31081,
 31082, 32708, 32737, 32936, 32938,
 32968, 33130, 33131, 33132, 33269,
 33290, 33334, 33349, 33379, 33433,
 33434, 33459, 33467, 33494, 33523,
 33544, 33551, 33583, 33584, 33590,
 33614, 33616, 33646, 33651, 33656,
 33674, 33679, 33694, 33699, 33819,
 33841, 33852, 33972, 35379, 35791,
 35795, 35796, 36337, 37629, 39212,
 39653, 39996, 41400, 42517, 42577,
 42625, 42630, 42660, 42666, 42684
- `\exp_stop_f`: 43,
 44, 185, 426, 458, 464, 675, 747,
 865, 881, 1064, 1077, 1149, 1150,
 1242, 1272, 1348, 2459, 2465, 3218,
 3478, 3672, 3681, 3682, 3692, 3703,
 3704, 3740, 3810, 3843, 3844, 3848,
 3925, 3941, 3947, 4033, 4196, 4499,
 4500, 4501, 4506, 4787, 4808, 4809,
 4813, 4817, 4818, 4821, 4822, 4837,
 4838, 4841, 4845, 4846, 4849, 4910,
 5261, 5266, 5280, 5281, 5294, 5356,
 5357, 5396, 6362, 6415, 6929, 6933,
 7081, 7105, 7140, 7145, 7151, 7496,
 8706, 8708, 8709, 8711, 9119, 10300,
 10543, 10846, 10860, 10872, 11082,
 13439, 13498, 13504, 14095, 14111,
 14151, 14157, 14169, 14186, 14324,
 14515, 14523, 14732, 14733, 14734,
 14739, 14740, 14764, 15135, 15197,
 15201, 15231, 15234, 15250, 15254,
 15276, 15356, 15358, 15378, 15379,
 15396, 15398, 15452, 15455, 15456,
 15575, 15580, 15721, 16633, 16816,
 17561, 18219, 18233, 18243, 18251,
 18460, 18465, 18654, 19869, 19871,
 19939, 19941, 19945, 19947, 19951,
 19953, 19957, 19959, 19999, 20000,
 20001, 20002, 20008, 20012, 20030,
 20139, 21174, 21724, 21895, 24039,
 24042, 24169, 24217, 24424, 24541,
 24556, 24819, 24845, 24894, 24906,
 24973, 25116, 25146, 25302, 25345,
 25396, 25416, 25443, 25457, 25493,
 25520, 25529, 25548, 25564, 25580,
 25598, 25659, 25678, 25694, 25932,
 26020, 26062, 26300, 26304, 26682,
 26684, 26699, 26716, 26724, 26725,
 26918, 27040, 27046, 27061, 27098,
 27173, 27196, 27251, 27252, 27260,
 27604, 27622, 27766, 27778, 27794,
 27811, 28102, 28103, 28202, 28297,
 28341, 28359, 28368, 28370, 28536,
 28571, 28724, 28776, 28822, 28827,
 28910, 28978, 28984, 28999, 29011,
 29049, 29070, 29110, 29125, 29140,
 29155, 29170, 29185, 29213, 29257,
 29523, 29533, 29563, 29742, 29744,
 29793, 29867, 29876, 29891, 29943,
 29956, 30043, 30097, 30148, 30171,
 30421, 30422, 30423, 30432, 30442,
 30460, 30532, 30535, 30538, 30677,
 30697, 30817, 31081, 31108, 31161,
 31165, 31207, 31279, 31286, 31375,
 32009, 32010, 32016, 32720, 32764,
 32881, 32888, 32905, 32908, 33116,
 34248, 34251, 34252, 34255, 34256,
 34277, 34280, 34283, 34286, 34289,
 34292, 34311, 34312, 34318, 34321,
 34324, 34327, 34330, 34333, 34336,
 34339, 34342, 34345, 34348, 34351,
 34354, 34357, 34360, 34389, 34392,
 34407, 34410, 34424, 34427, 34430,
 34433, 34449, 34452, 34455, 41997
- `exp` internal commands:
- `\__exp_arg_last_unbraced:nn`
 .. 2633, 2633, 2635, 2638, 2643, 2650
- `\__exp_arg_next:Nnn` . 2447, 2448, 2454
- `\__exp_arg_next:nnn`
 426, 2447, 2447, 2456, 2461, 2474, 2480
- `\__exp_eval_error_msg:w`
 2484, 2488, 2497
- `\__exp_eval_register:N` 2475,
 2481, 2484, 2484, 2495, 2496, 2527,
 2532, 2538, 2544, 2568, 2574, 2586,
 2587, 2594, 2601, 2639, 2644, 2660,
 2662, 2673, 2687, 2696, 2735, 2740
- `\__exp_last_two_unbraced:nnN` ...
 2721, 2722, 2723
- `\l__exp_tmp_tl` 395, 1485, 1489, 1490,
 2447, 2447, 2468, 2470, 2655, 2656
- `\expandafter` 3, 4, 7,
 8, 12, 13, 16, 17, 28, 29, 53, 54, 61, 228
- `\expanded` 820
- `\expandglyphsinfont` 935
- `\ExplFileDate` 11, 11663, 11678, 11692, 11696
- `\ExplFileDescription` ... 11, 11662, 11675
- `\ExplFileExtension` .. 11665, 11680, 11689
- `\ExplFileName` ... 11, 11664, 11679, 11688

- `\ExplFileVersion` 11, 11666, 11681, 11690
`\explicitdiscretionary` 821
`\explicitthyphenpenalty` 819
`\ExplSyntaxOff` 6,
10, 190, 353, 354, 383, 82, 110, 123
`\ExplSyntaxOn` 6, 10,
190, 296, 353, 354, 383, 721, 932, 106
- F**
- `fact` 282
`false` 287
`\fam` 229
`\fi` 6, 15, 20, 32, 33, 34, 54, 56, 57, 72, 80, 230
fi commands:
`\fi:` 29, 67, 74, 102,
185, 186, 214, 244, 323, 398, 400,
402, 405, 406, 464, 486, 487, 584,
585, 588, 615, 684, 721, 728, 769,
771, 773, 887, 898, 935, 944, 967,
968, 977, 978, 1077, 1105, 1120,
1155, 1386, 1390, 1433, 1655, 1663,
1671, 1679, 1699, 1704, 1717, 1725,
1733, 1735, 1736, 1737, 1738, 1763,
1768, 1775, 1802, 1807, 1829, 1834,
1845, 1848, 1855, 1861, 1863, 1864,
1871, 1876, 1884, 1889, 1891, 1892,
1899, 1905, 1907, 1908, 1924, 1927,
1928, 1935, 1938, 1939, 1945, 2065,
2086, 2096, 2110, 2168, 2253, 2432,
2442, 2489, 2492, 2499, 2500, 2749,
2788, 2804, 2811, 2820, 2834, 2835,
2840, 2841, 2842, 2860, 2861, 2862,
2863, 2864, 2865, 2866, 2867, 2868,
2876, 2895, 2897, 2927, 2928, 2929,
2976, 3064, 3075, 3085, 3220, 3231,
3232, 3276, 3307, 3350, 3361, 3370,
3379, 3434, 3444, 3454, 3566, 3568,
3645, 3650, 3654, 3684, 3695, 3707,
3708, 3719, 3737, 3738, 3739, 3746,
3762, 3768, 3771, 3779, 3789, 3804,
3812, 3820, 3831, 3847, 3867, 3881,
3903, 3926, 3934, 3936, 3939, 3946,
3951, 4039, 4040, 4100, 4136, 4137,
4138, 4141, 4150, 4173, 4184, 4219,
4220, 4230, 4243, 4286, 4287, 4294,
4295, 4300, 4301, 4326, 4330, 4405,
4462, 4469, 4470, 4476, 4480, 4487,
4488, 4493, 4494, 4503, 4504, 4508,
4509, 4523, 4531, 4540, 4541, 4568,
4724, 4732, 4743, 4749, 4761, 4800,
4801, 4811, 4814, 4815, 4819, 4823,
4824, 4825, 4826, 4831, 4842, 4843,
4847, 4850, 4851, 4852, 4857, 4891,
4892, 4912, 4913, 4922, 4930, 4931,
4941, 4942, 4951, 4961, 4962, 4973,
4974, 4987, 5006, 5007, 5015, 5016,
5081, 5091, 5124, 5138, 5142, 5205,
5250, 5253, 5254, 5270, 5273, 5296,
5354, 5355, 5360, 5387, 5388, 5399,
5403, 5437, 5442, 5450, 5485, 5492,
5497, 5507, 5519, 5545, 5608, 5637,
5676, 5683, 5694, 5782, 5801, 5807,
5812, 5835, 5847, 5848, 5851, 5863,
5897, 5927, 6311, 6350, 6366, 6390,
6410, 6419, 6476, 6483, 6503, 6521,
6532, 6534, 6564, 6567, 6601, 6607,
6613, 6710, 6747, 6778, 6779, 6809,
6835, 6880, 6932, 6978, 6979, 6991,
7042, 7044, 7097, 7109, 7119, 7128,
7148, 7159, 7185, 7201, 7209, 7259,
7261, 7276, 7278, 7299, 7478, 7499,
7577, 7594, 7595, 7615, 7654, 7656,
7662, 7664, 7669, 7699, 7722, 7738,
7824, 7826, 7827, 7833, 8012, 8028,
8421, 8472, 8491, 8513, 8533, 8549,
8559, 8575, 8585, 8698, 8700, 8702,
8704, 8708, 8711, 8978, 8986, 10402,
10405, 10408, 10485, 10503, 10797,
10839, 10848, 10869, 10879, 10883,
10890, 10898, 11093, 11097, 11100,
11150, 11163, 11190, 11199, 11460,
11469, 11480, 12221, 12485, 12492,
12721, 12722, 12782, 12792, 12807,
12816, 12835, 12849, 12862, 12866,
12879, 12897, 12912, 13151, 13156,
13181, 13199, 13219, 13227, 13237,
13247, 13253, 13260, 13271, 13276,
13278, 13282, 13287, 13291, 13292,
13441, 13453, 13502, 13508, 13509,
13613, 13620, 13625, 13662, 13669,
13675, 13679, 13822, 13827, 13832,
13839, 13844, 13945, 14023, 14027,
14028, 14046, 14100, 14113, 14155,
14161, 14162, 14173, 14186, 14187,
14208, 14248, 14274, 14385, 14492,
14500, 14508, 14519, 14536, 14538,
14684, 14736, 14737, 14742, 14745,
14746, 14768, 14821, 14921, 15137,
15170, 15206, 15207, 15238, 15239,
15259, 15260, 15279, 15364, 15368,
15378, 15388, 15404, 15408, 15411,
15416, 15418, 15461, 15465, 15466,
15557, 15562, 15573, 15579, 15591,
15594, 15596, 15600, 15714, 15728,
15729, 16634, 16649, 17072, 17080,
17104, 17118, 17126, 17137, 17147,
17167, 17197, 17198, 17270, 17296,
17307, 17326, 17329, 17602, 17605,

17674, 17710, 17763, 17780, 17790,
17844, 17849, 18226, 18227, 18236,
18259, 18276, 18277, 18279, 18296,
18297, 18341, 18418, 18426, 18453,
18461, 18467, 18490, 18522, 18560,
18568, 18640, 18656, 18876, 18909,
18957, 18962, 19078, 19105, 19132,
19141, 19345, 19360, 19383, 19397,
19999, 20000, 20001, 20002, 20007,
20008, 20016, 20017, 20018, 20047,
20053, 20071, 20080, 20082, 20128,
20129, 20130, 20131, 20132, 20133,
20134, 20135, 20136, 20137, 20166,
20171, 20176, 20181, 20188, 20195,
20200, 20205, 20210, 20215, 20220,
20227, 20232, 20254, 20265, 20266,
20316, 20317, 20340, 20343, 20348,
20350, 20356, 20361, 20363, 20364,
20467, 20476, 20485, 20493, 20570,
20599, 20600, 20608, 20646, 20660,
20669, 20679, 20701, 20702, 20703,
20713, 20720, 20722, 21046, 21053,
21097, 21145, 21154, 21302, 21341,
21350, 21381, 21382, 21383, 21384,
21393, 21394, 21395, 21396, 21419,
21420, 21421, 21422, 21431, 21432,
21433, 21434, 21650, 21673, 21682,
21723, 21727, 21741, 21744, 21906,
21907, 21950, 21965, 21966, 22505,
22572, 22584, 24044, 24045, 24080,
24088, 24152, 24171, 24185, 24273,
24289, 24293, 24305, 24315, 24410,
24464, 24467, 24468, 24473, 24487,
24525, 24526, 24527, 24528, 24529,
24530, 24531, 24532, 24534, 24535,
24536, 24537, 24550, 24552, 24563,
24566, 24580, 24585, 24589, 24724,
24825, 24826, 24835, 24836, 24847,
24848, 24849, 24860, 24861, 24862,
24869, 24880, 24881, 24882, 24892,
24893, 24897, 24898, 24906, 24909,
24910, 24918, 24929, 24949, 24973,
25010, 25027, 25046, 25047, 25056,
25062, 25082, 25083, 25111, 25120,
25137, 25144, 25152, 25153, 25254,
25255, 25256, 25259, 25262, 25301,
25317, 25343, 25344, 25351, 25359,
25388, 25389, 25392, 25394, 25395,
25400, 25410, 25413, 25415, 25420,
25451, 25464, 25470, 25476, 25479,
25480, 25514, 25515, 25542, 25543,
25556, 25559, 25570, 25593, 25612,
25622, 25638, 25647, 25653, 25659,
25663, 25668, 25674, 25689, 25700,
25717, 25725, 25727, 25733, 25754,
25782, 25805, 25838, 25840, 25967,
26015, 26019, 26029, 26030, 26046,
26057, 26061, 26071, 26072, 26092,
26113, 26116, 26146, 26178, 26194,
26214, 26255, 26267, 26280, 26282,
26302, 26303, 26310, 26328, 26367,
26377, 26595, 26611, 26622, 26662,
26663, 26664, 26671, 26673, 26674,
26680, 26681, 26684, 26711, 26719,
26720, 26728, 26729, 26731, 26732,
26899, 26912, 26922, 26923, 26929,
26930, 26931, 26932, 26933, 26934,
26941, 26951, 26958, 26970, 26971,
26982, 26999, 27044, 27045, 27052,
27065, 27080, 27090, 27104, 27134,
27143, 27179, 27202, 27220, 27237,
27255, 27256, 27258, 27259, 27264,
27279, 27313, 27342, 27343, 27344,
27345, 27346, 27359, 27404, 27480,
27549, 27551, 27552, 27562, 27591,
27594, 27595, 27606, 27626, 27689,
27690, 27691, 27703, 27742, 27743,
27744, 27745, 27751, 27754, 27756,
27766, 27784, 27799, 27811, 27818,
28076, 28080, 28082, 28086, 28093,
28094, 28106, 28107, 28110, 28204,
28276, 28287, 28299, 28340, 28347,
28358, 28374, 28381, 28444, 28508,
28519, 28521, 28531, 28549, 28550,
28582, 28585, 28594, 28596, 28598,
28612, 28626, 28650, 28734, 28742,
28775, 28783, 28789, 28800, 28803,
28806, 28815, 28824, 28826, 28832,
28839, 28842, 28851, 28859, 28881,
28914, 28915, 28942, 28944, 28963,
28964, 28983, 28994, 29003, 29006,
29017, 29020, 29023, 29041, 29051,
29062, 29064, 29073, 29120, 29135,
29150, 29165, 29180, 29195, 29198,
29200, 29217, 29262, 29569, 29605,
29606, 29616, 29674, 29675, 29709,
29736, 29737, 29740, 29742, 29743,
29748, 29760, 29779, 29784, 29792,
29795, 29827, 29837, 29838, 29849,
29871, 29886, 29904, 29912, 29915,
29943, 29951, 29967, 30042, 30060,
30096, 30114, 30147, 30170, 30176,
30249, 30250, 30355, 30356, 30365,
30372, 30377, 30387, 30397, 30422,
30425, 30438, 30470, 30478, 30479,
30507, 30532, 30533, 30534, 30537,
30542, 30562, 30563, 31093, 31096,
31170, 31171, 31212, 31220, 31273,

- 31279, 31292, 31373, 32044, 32045,
32048, 32457, 32501, 32505, 32506,
32534, 32694, 32698, 32709, 32724,
32736, 32740, 32756, 32768, 32800,
32801, 32802, 32803, 32804, 32805,
32806, 32896, 32900, 32914, 32915,
33319, 34260, 34263, 34264, 34267,
34268, 34296, 34297, 34298, 34299,
34300, 34301, 34316, 34364, 34365,
34366, 34367, 34368, 34369, 34370,
34371, 34372, 34373, 34374, 34375,
34376, 34377, 34378, 34379, 34396,
34397, 34414, 34415, 34437, 34438,
34439, 34440, 34459, 34460, 34461,
36478, 36480, 36486, 41863, 41874,
41875, 41955, 41956, 41957, 41958,
41971, 41972, 41988, 41989, 41990,
41991, 41992, 41993, 41994, 41995,
41996, 42757, 42758, 42763, 42764
- file commands:
- \file_compare_timestamp:nNn
..... 11442, 11450
 - \file_compare_timestamp:nNnTF ...
..... 105, 11442
 - \file_compare_timestamp_p:nNn ...
..... 105, 11442
 - \g_file_curr_dir_str
.... 102, 11004, 11535, 11541, 11554
 - \g_file_curr_ext_str
.... 102, 11004, 11537, 11543, 11556
 - \g_file_curr_name_str 102,
9234, 9356, 11004, 11536, 11542, 11555
 - \file_forget:n 103, 11281, 11281
 - \file_full_name:n 105,
11173, 11173, 11178, 11282, 11292,
11309, 11317, 11324, 11371, 11446,
11447, 11487, 11559, 40216, 40221
 - \file_get:nnN
106, 11130, 11130, 11135, 11136, 11147
 - \file_get:nnNTF ... 106, 11130, 11132
 - \file_get_full_name:nN
105, 11283, 11283, 11288, 11289, 11297
 - \file_get_full_name:nNTF
..... 105, 10291,
11138, 11283, 11285, 11494, 11500,
11513, 40601, 40711, 40738, 40748
 - \file_get_hex_dump:nN
104, 11388, 11388, 11390, 11400, 11402
 - \file_get_hex_dump:nnnN
104, 11424, 11424, 11429, 11430, 11439
 - \file_get_hex_dump:nnnNTF
..... 104, 11424, 11426
 - \file_get_hex_dump:nNTF
..... 104, 11388, 11389
 - \file_get_md5five_hash:nN
104, 11388, 11391, 11393, 11404, 11406
 - \file_get_md5five_hash:nNTF
..... 104, 11388, 11392
 - \file_get_size:nN
104, 11388, 11394, 11396, 11408, 11410
 - \file_get_size:nNTF 104, 11388, 11395
 - \file_get_timestamp:nN
104, 11388, 11397, 11399, 11412, 11414
 - \file_get_timestamp:nNTF
..... 104, 11388, 11398
 - \file_hex_dump:n
..... 103, 104, 11321, 11370, 11372
 - \file_hex_dump:nnn 103,
104, 11321, 11321, 11328, 11434, 40198
 - \file_if_exist:n 11485, 11491
 - \file_if_exist:nTF 103, 105,
106, 11485, 11812, 11814, 11818, 14703
 - \file_if_exist_input:n
..... 106, 11492, 11492, 11497
 - \file_if_exist_input:nTF
..... 106, 11492, 11498, 11504
 - \file_if_exist_p:n 103, 11485
 - \file_input:n
106, 107, 11511, 11511, 11517, 14707
 - \file_input_raw:n
..... 106, 11558, 11558, 11560
 - \file_input_stop: .. 107, 11505, 11505
 - \file_log_list: 107, 341, 11631, 11632
 - \file_md5five_hash:n
..... 104, 11300, 11316, 11318
 - \file_parse_full_name:n
..... 106, 695, 11572, 11572, 11577
 - \file_parse_full_name:nnNN
..... 105, 106, 11539,
11619, 11619, 11630, 40615, 40716
 - \file_parse_full_name_apply:nN ..
..... 106,
694, 11572, 11574, 11578, 11583, 11621
 - \l_file_search_path_seq 103, 104,
106, 11038, 11205, 40600, 40710, 40747
 - \file_show_list: 107, 341, 11631, 11631
 - \file_size:n
..... 103, 104, 11300, 11300, 11302
 - \file_timestamp:n
..... 76, 104, 11300, 11303, 11305
- file internal commands:
- \l_file_base_name_tl 11033
 - __file_compare_timestamp:nnN ...
..... 11442, 11445, 11452
 - __file_const:nn 11826
 - __file_details:nn
..... 11300, 11301, 11304, 11306

```

\__file_details_aux:nn .....
    ..... 11300, 11308, 11311, 11339
\l__file_dir_str . 11035, 11540, 11541
\__file_ext_check:nn .....
    ..... 11214, 11240, 11247
\__file_ext_check:nnn . 11262, 11267
\__file_ext_check:nnnn . 11268, 11269
\__file_ext_check:nnnw . 11253, 11258
\__file_ext_check:nnw .....
    ..... 11248, 11249, 11256
\l__file_ext_str . 11035, 11540, 11543
\__file_full_name:n .....
    ..... 11173, 11175, 11179
\__file_full_name_assign:nnnNNN .
    ..... 11622, 11624
\__file_full_name_aux:n .....
    .. 11173, 11182, 11184, 11233, 11268
\__file_full_name_aux:nN .....
    ..... 11173, 11218, 11232
\__file_full_name_aux:Nnn .....
    ..... 11173, 11206, 11210, 11216
\__file_full_name_aux:nnN .....
    ..... 11173, 11233, 11234
\__file_full_name_auxi:nn .....
    ..... 11173, 11189, 11192
\__file_full_name_auxii:nn .....
    ..... 11173, 11182, 11201
\__file_full_name_slash:n .....
    ..... 11173, 11219, 11222
\__file_full_name_slash:nw .....
    ..... 11224, 11226
\__file_full_name_slash:w .... 11173
\l__file_full_name_tl .....
    .... 11033, 11138, 11141, 11494,
    11495, 11500, 11501, 11513, 11514
\__file_get_aux:nnN .....
    ..... 11130, 11140, 11148
\__file_get_details:nnN .. 11388,
    11401, 11405, 11409, 11413, 11416
\__file_get_do:Nw 11130, 11156, 11166
\__file_get_full_name_search:nN .
    ..... 11283
\__file_hex_dump:n .....
    ..... 11321, 11371, 11375, 11382
\__file_hex_dump_auxi:nnn .....
    ..... 11321, 11323, 11329
\__file_hex_dump_auxii:nnnn ....
    ..... 11321, 11338, 11343
\__file_hex_dump_auxiii:nnnn ...
    ..... 11321, 11346, 11348, 11353
\__file_hex_dump_auxiiv:nnn .. 11321
\__file_hex_dump_auxiv:nnn .....
    ..... 11356, 11358, 11363

\__file_id_info_auxi:w .....
    ..... 11660, 11671, 11673
\__file_id_info_auxii:w .....
    ..... 697, 11660, 11683, 11685
\__file_id_info_auxiii:w .....
    ..... 11660, 11693, 11695
\__file_if_recursion_tail_-
    break:NN ..... 11045
\__file_if_recursion_tail_stop:N
    ..... 11045
\__file_if_recursion_tail_stop_-
    do:Nn ..... 11045
\__file_if_recursion_tail_stop_-
    do:nn ..... 11046
\__file_input:n ..... 11495,
    11501, 11511, 11514, 11518, 11530
\__file_input_pop: .....
    ..... 11511, 11528, 11546, 11551
\__file_input_pop:nnn .....
    ..... 11511, 11549, 11552
\__file_input_push:n .....
    ..... 11511, 11523, 11531, 11545
\__file_input_raw:nn .....
    ..... 11558, 11559, 11561
\__file_kernel_dependency_-
    compare:nnn .....
    ..... 11697, 11703, 11706, 11708
\__file_list:N .....
    ..... 11631, 11631, 11632, 11633
\__file_list_aux:n 11631, 11644, 11647
\c__file_marker_tl .....
    ..... 684, 11129, 11154, 11167
\__file_md5five_hash:n .....
    ..... 11300, 11317, 11319
\__file_mismatched_dependency_-
    error:nn ..... 11713, 11716, 11716
\__file_name_cleanup:w .....
    ..... 11173, 11241, 11245
\__file_name_end: .....
    ..... 11173, 11212, 11245, 11246
\__file_name_expand:n .....
    ..... 11047, 11052, 11055
\__file_name_expand_cleanup:Nw ..
    ..... 682, 11047, 11057, 11061
\__file_name_expand_cleanup:w ...
    ..... 682, 11047, 11065, 11068
\__file_name_expand_end: .....
    . 682, 11047, 11059, 11061, 11064,
    11069, 11073, 11075, 11076, 11078
\__file_name_expand_error:Nw ...
    ..... 682, 683, 11047, 11064, 11075
\__file_name_expand_error_aux:Nw
    ..... 683, 11047, 11076, 11077
\__file_name_ext_check:nn .... 11173

```

__file_name_ext_check:nnn ...	11173	\filemoddate	771
__file_name_ext_check:nnnn ..	11173	\filesize	772
__file_name_ext_check:nnnw ..	11173	\finalhyphendemerits	231
__file_name_ext_check:nnw ...	11173	\firstmark	232
__file_name_quote:nw		\firstmarks	489
.....	11121 , 11122 , 11123	\firstvalidlanguage	822
\l__file_name_str	11035 , 11540 , 11542	\fixupboxesmode	823
__file_name_strip_quotes:n		flag commands:	
.....	11047 , 11051 , 11084	\flag_clear:N ...	188 , 5712 , 7600 ,
__file_name_strip_quotes:nnn .	11047	7601 , 14781 , 14809 , 14903 , 14932 ,	
__file_name_strip_quotes:nnnw	11047	15001 , 15049 , 15050 , 15102 , 15103 ,	
__file_name_strip_quotes:nw ...		15340 , 15341 , 15342 , 15343 , 15344 ,	
.....	11086 , 11089 , 11095 , 11098	15445 , 15540 , 15541 , 15542 , 15543 ,	
__file_name_strip_quotes_-		15698 , 15699 , 15700 , 19095 , 19095 ,	
end:wnwn	11092 , 11097	19100 , 19111 , 19154 , 30917 , 42345	
__file_name_trim_spaces:n		\flag_clear:n	19153 , 19154
.....	11047 , 11049 , 11107	\flag_clear_new:N	
__file_name_trim_spaces:nw		..	188 , 803 , 15285 , 15286 , 15287 ,
.....	11047 , 11108 , 11109	15288 , 15468 , 15469 , 15470 , 15648 ,	
__file_name_trim_spaces_aux:n ..		15649 , 19110 , 19110 , 19112 , 19155	
.....	11047 , 11114 , 11118	\flag_clear_new:n	19153 , 19155
__file_name_trim_spaces_aux:w ..		\flag_ensure_raised:N	
.....	11047 , 11119 , 11120	..	188 , 5739 , 5761 , 19150 , 19150 ,
__file_parse_full_name_area:nw .		19152 , 19166 , 24648 , 24659 , 24667 ,	
....	695 , 11584 , 11586 , 11589 , 11593	24684 , 24697 , 24728 , 30916 , 42319	
__file_parse_full_name_auxi:nN .		\flag_ensure_raised:n .	19153 , 19166
.....	11580 , 11584 , 11584	\flag_height:N	188 ,
__file_parse_full_name_base:nw .		7610 , 7612 , 14625 , 19120 , 19136 ,	
....	695 , 11592 , 11595 , 11595 , 11607	19136 , 19146 , 19148 , 19164 , 42320	
__file_parse_full_name_tidy:nnnN		\flag_height:n ..	19153 , 19164 , 19174
695 , 11602 , 11603 , 11605 , 11609 , 11609		\flag_if_exist:N	19122 , 19124
__file_parse_version:w		\flag_if_exist:NTF ...	188 , 19111 ,
.....	11697 , 11711 , 11712 , 11715	19122 , 19157 , 19158 , 19159 , 41817	
__file_quark_if_nil:n	11042	\flag_if_exist:nTF	
__file_quark_if_nil:nTF			19153 , 19157 , 19158 , 19159
..	11042 , 11111 , 11125 , 11251 , 11260	\flag_if_exist_p:N	
__file_quark_if_nil_p:n	11042		188 , 381 , 19122 , 19156
\g__file_record_seq ...	693 , 696 ,	\flag_if_exist_p:n	19153
697 , 11032 , 11522 , 11641 , 11655 , 11656		\flag_if_raised:N	19126 , 19134
__file_size:n ..	11172 , 11172 , 11189	\flag_if_raised:NTF ...	188 , 5720 ,
\g__file_stack_seq		14618 , 14623 , 14625 , 15312 , 15318 ,	
.....	693 , 11007 , 11533 , 11548	15323 , 15330 , 15502 , 15507 , 15512 ,	
__file_str_cmp:nn	11441 , 11441 , 11473	15663 , 15670 , 19126 , 19161 , 19162 ,	
__file_timestamp:n		19163 , 30919 , 42321 , 42322 , 42323	
.....	11442 , 11474 , 11475 , 11484	\flag_if_raised:nTF	
__file_tmp:w			19153 , 19161 , 19162 , 19163
..	11009 , 11013 , 11017 , 11023 , 11029	\flag_if_raised_p:N	
\g__file_tmp_ior			188 , 19126 , 19160 , 42324
.....	11299 , 11718 , 11729 , 11731	\flag_if_raised_p:n	19153
\l__file_tmp_seq ...	11039 , 11635 ,	\flag_log:N ..	188 , 19113 , 19115 , 19116
11638 , 11641 , 11642 , 11644 , 11652 ,		\flag_log:n	19167 , 19168
11657 , 11728 , 11730 , 11749 , 11753		\flag_new:N	
\l__file_tmp_tl .	11003 , 11548 , 11549	..	187 , 188 , 803 , 5702 , 7460 , 7461 ,
\filedump	770	14487 , 14488 , 19090 , 19090 , 19092 ,	

- 19093, 19094, 19111, 19153, 24614,
24615, 24616, 24617, 30899, 42338
- \flag_new:n 19153, 19153
- \flag_raise:N 188,
7653, 7661, 14767, 14817, 14917,
14950, 15021, 15034, 15072, 15077,
15158, 15361, 15362, 15385, 15386,
15399, 15400, 15419, 15420, 15426,
15427, 15457, 15607, 15608, 15717,
15718, 15722, 15723, 15737, 15738,
19147, 19147, 19149, 19165, 42325
- \flag_raise:n 19153, 19165
- \flag_show:N . 188, 19113, 19113, 19114
- \flag_show:n 19167, 19167
- \l_tmpa_flag .. 189, 381, 19093, 41821
- \l_tmpb_flag 189, 19093
- flag internal commands:
- _flag_clear:wN
 19095, 19097, 19101, 19107
- _flag_height_end:wN
 19136, 19140, 19145
- _flag_height_loop:wN
 19136, 19136, 19137, 19142
- _flag_sep:
 19089, 19089, 19097, 19101,
 19108, 19136, 19137, 19143, 19145
- _flag_show:NN
 19113, 19113, 19115, 19117
- _flag_show:Nn
 19167, 19167, 19168, 19169
- \floatingpenalty 233
- floor 283
- \fmtname 8776, 8779, 8780,
8792, 8802, 32948, 33211, 33212,
37502, 37503, 38405, 38406, 40875
- \font 234
- \fontcharhp 490
- \fontcharht 491
- \fontcharic 492
- \fontcharwd 493
- \fontdimen 1049, 235
- \fontencoding 35720
- \fontfamily 35721
- \fontid 825
- \fontname 236
- \fontseries 35722
- \fontshape 35723
- \fontsize 35726
- \forcecjktoken 1202
- \formatname 826
- fp commands:
- _c_e_fp 276, 279, 26572
- \fp_abs:n 281,
287, 1266, 30271, 30271, 36982,
37084, 37086, 37088, 38114, 38116
- \fp_add:Nn
 267, 1266, 26480, 26480, 26486
- \fp_clear_function:n 275, 31110, 31110
- \fp_clear_variable:n
 275, 30865, 30865, 31049
- \fp_compare:n 26597
- \fp_compare:nNn 26613
- \fp_compare:nNnTF
 ... 270–272, 26613, 26765, 26771,
 26776, 26784, 26841, 26847, 36839,
 36841, 36846, 37116, 37131, 37140,
 37856, 38088, 38840, 39023, 39027,
 39035, 39042, 39049, 39056, 39063,
 39110, 39130, 39572, 39575, 40021
- \fp_compare:nTF
 ... 270–272, 281, 17026, 26597,
 26737, 26743, 26748, 26756, 41316
- \fp_compare_p:n 271, 26597
- \fp_compare_p:nNn 270, 26613, 38816,
38817, 38836, 38837, 40853, 40854
- \fp_const:Nn ... 267, 26457, 26461,
26465, 26572, 26573, 26574, 26575
- \l_fp_division_by_zero_flag
 277, 24614, 24684, 24697
- \fp_do_until:nm
 272, 26734, 26734, 26738
- \fp_do_until:nNnn
 271, 26762, 26762, 26766
- \fp_do_while:nm
 272, 26734, 26740, 26744
- \fp_do_while:nNnn
 272, 26762, 26768, 26772
- \fp_eval:n
 268, 271, 275, 280–287, 295, 1142,
 1303, 30266, 30268, 31433, 38707,
 38709, 38715, 38716, 38717, 38722,
 38726, 38727, 38728, 38732, 38735,
 38736, 38737, 38897, 38907, 38911,
 38912, 38913, 38918, 38919, 38920,
 38921, 38949, 38954, 38962, 38968,
 38977, 38978, 38979, 38995, 38996,
 38997, 39005, 39006, 39007, 39014,
 39015, 39016, 39078, 39083, 39114,
 39131, 39132, 39137, 39138, 39139,
 39144, 39145, 39146, 39151, 39152,
 39153, 39161, 39162, 39724, 39725,
 39726, 39727, 39739, 39740, 39741,
 39752, 39877, 39878, 39940, 40099,
 40100, 40101, 40102, 40110, 40111,
 40112, 40113, 40129, 40135, 40152,
 40153, 40154, 40162, 40163, 40164

- \fp_format:nn 150, 270, 288, 16891, 16891
- \fp_gadd:Nn .. 267, 26480, 26481, 26487
- .fp_gset:N 248, 23014
- \fp_gset:Nn 267, 26457, 26459, 26464, 26481, 26483, 41174, 41244, 41245, 41257, 41297, 41310, 41356
- \fp_gset_eq:NN 267, 26466, 26467, 26469, 26471, 42106, 42236
- \fp_gsub:Nn .. 267, 26480, 26483, 26489
- \fp_gzero:N 267, 26470, 26471, 26473, 26477
- \fp_gzero_new:N 267, 26474, 26476, 26479
- \fp_if_exist:N 26587, 26588
- \fp_if_exist:NTF 270, 26475, 26477, 26587, 30813
- \fp_if_exist_p:N 270, 26587
- \fp_if_nan:n 26589
- \fp_if_nan:nTF 271, 288, 26589
- \fp_if_nan_p:n 271, 26589
- \l_fp_invalid_operation_flag ... 277, 24614, 24648, 24659, 24667
- \fp_log:N 278, 26490, 26492, 26493
- \fp_log:n 278, 26568, 26570
- \fp_max:nn 287, 30273, 30273
- \fp_min:nn 287, 30273, 30275
- \fp_new:N 267, 26454, 26454, 26456, 26475, 26477, 26576, 26577, 26578, 26579, 30627, 36805, 36806, 36807, 36933, 36934, 37374, 37375, 37883, 37884, 38048, 38049, 40821, 41173, 41225, 41226, 41302
- \fp_new_function:n 274, 275, 30962, 30962
- \fp_new_variable:n 273–275, 1288, 30880, 30880
- \l_fp_overflow_flag 277, 24614
- .fp_set:N 248, 23014
- \fp_set:Nn 267, 273, 26457, 26457, 26463, 26480, 26482, 30914, 30918, 31070, 36827, 36828, 36829, 36952, 36954, 36995, 37015, 37035, 37052, 37054, 37072, 37073, 37113, 37114, 37901, 37902, 38068, 38070, 38108, 38109, 40851
- \fp_set_eq:NN 267, 26466, 26466, 26468, 26470, 37000, 37020, 37037, 37117, 37118, 42105, 42154
- \fp_set_function:nnn 275, 1292, 31025, 31025
- \fp_set_variable:nn 273–275, 1286, 1288, 30899, 30900
- \fp_show:N 273, 274, 278, 26490, 26490, 26491
- \fp_show:n 273–275, 278, 1288, 26568, 26568
- \fp_sign:n 268, 30269, 30269
- \fp_step_function:nnnN 273, 26790, 26790, 26797, 26878
- \fp_step_inline:nnnn 273, 26856, 26856
- \fp_step_variable:nnnN 273, 26856, 26863
- \fp_sub:Nn 267, 26480, 26482, 26488
- \fp_to_decimal:N . 268, 269, 24605, 30073, 30073, 30075, 30104, 30266
- \fp_to_decimal:n 268, 269, 844, 17000, 17027, 30073, 30076, 30268, 30270, 30272, 30274, 30276
- \fp_to_dim:N 268, 1264, 30196, 30196, 30198
- \fp_to_dim:n 268, 277, 30196, 30199, 36871, 36882, 36982, 37811, 37833, 37861, 37875, 37985, 37993, 38124, 38126
- \fp_to_int:N . 268, 30212, 30212, 30213
- \fp_to_int:n ... 268, 30212, 30214, 39576, 41276, 41306, 41312, 41320
- \fp_to_scientific:N 269, 30019, 30019, 30021, 30050, 30057
- \fp_to_scientific:n 269, 843, 844, 16966, 17030, 30019, 30022
- \fp_to_tl:N . 269, 291, 1549, 24606, 26498, 30152, 30152, 30153, 30924
- \fp_to_tl:n 269, 16892, 16950, 16988, 17010, 24230, 24647, 24656, 24657, 24683, 24694, 24695, 24725, 26340, 26355, 26569, 26571, 26808, 26809, 26829, 26844, 30152, 30154, 41317, 41324
- \fp_trap:nn 277, 1082, 24618, 24618, 24739, 24740, 24741, 24742
- \l_fp_underflow_flag 277, 24614
- \fp_until_do:nn 272, 26734, 26746, 26751
- \fp_until_do:nNnn 272, 26762, 26774, 26779
- \fp_use:N 269, 291, 30266, 30266, 30267, 40857
- \fp_while_do:nn 272, 26734, 26754, 26759
- \fp_while_do:nNnn 272, 26762, 26782, 26787
- \fp_zero:N 267, 26470, 26470, 26472, 26475
- \fp_zero_new:N 267, 26474, 26474, 26478

- \c_inf_fp 276,
286, [24244](#), 25848, 27356, 27441,
27782, 28578, 28601, 28805, 28808,
28812, 28835, 29039, 29202, 31290
- \c_minus_inf_fp
..... 276, 286, [24244](#), 27357,
27444, 27780, 28343, 29203, 31291
- \c_minus_zero_fp
..... 276, [24244](#), 27353, 29950, 31289
- \c_nan_fp
276, 286, 1085, 1110, [24244](#), 24660,
24668, 24744, 24960, 24981, 24988,
25011, 25178, 25186, 25194, 25272,
25329, 25368, 25760, 25837, 25849,
26342, 26357, 26834, 28779, 30330,
30916, 31104, 31203, 31262, 31288
- \c_one_degree_fp 276, 286, 25851, [26574](#)
- \c_one_fp 276, 1138, 1249,
25852, 26285, 26306, [26572](#), 26938,
27803, 28572, 28774, 28825, 29012,
29126, 29156, 29732, 30346, 40854
- \c_pi_fp . 276, 286, 1120, 25850, [26574](#)
- \g_tmpa_fp 276, [26576](#)
- \l_tmpa_fp 276, [26576](#)
- \g_tmpb_fp 276, [26576](#)
- \l_tmpb_fp 273, 274, 276, [26576](#)
- \c_zero_fp 276, 1142, 1160,
[1295](#), [24244](#), 24298, 25853, 26297,
26309, 26455, 26470, 26471, 26940,
26943, 27183, 27352, 28581, 28602,
28802, 28838, 29948, 30057, 30241,
31287, 36839, 36841, 36846, 37131,
37140, 38088, 38840, 40021, 40853
- fp internal commands:
- __fp_ 26947, 26954, 26963, 26964
- __fp_&o:ww 1146, 1155, [26944](#)
- __fp_&symbolic_o:ww [30681](#)
- __fp_&tuple_o:ww [26944](#)
- __fp_*_o:ww [27317](#)
- __fp_*_symbolic_o:ww [30681](#)
- __fp_*_tuple_o:ww [27830](#)
- __fp+_o:ww . 1158, 1159, 1188, [27033](#)
- __fp+_symbolic_o:ww [30681](#)
- __fp_-o:ww 1158, 1159, [27028](#)
- __fp_-symbolic_o:ww [30681](#)
- __fp_/o:ww 1168, 1211, [27432](#)
- __fp_/symbolic_o:ww [30681](#)
- __fp_{op}_o:w 1280
- __fp^_o:ww [28770](#)
- __fp^_symbolic_o:ww [30681](#)
- __fp_acos_o:w 1254, 1256, 29889, 29889
- __fp_acot_o:Nw
..... 29101, 29103, 29720, 29726
- __fp_acotii_o:Nww 29730, [29733](#), 29753
- __fp_acotii_o:ww 1250
- __fp_acsc_normal_o:NnwNnw
... 1256, 29947, 29962, [29970](#), 29970
- __fp_acsc_o:w [29941](#), 29941
- __fp_add:NNNn [26480](#),
26480, 26481, 26482, 26483, 26484
- __fp_add_big_i:wNww 1161
- __fp_add_big_i_o:wNww
..... 1158, 1161, 27100, [27107](#), 27107
- __fp_add_big_ii:wNww 1161
- __fp_add_big_ii_o:wNww
..... 27103, [27107](#), 27115
- __fp_add_inf_o:Nww
..... 27049, [27069](#), 27069
- __fp_add_normal_o:Nww
..... 1161, 27048, [27084](#), 27084
- __fp_add_npos_o:NnwNnw
..... 1161, 27087, [27093](#), 27093
- __fp_add_return_ii_o:Nww
..... 27051, [27057](#), 27057, 27062
- __fp_add_significand_carry_-
o:wwwNN . 1163, 27140, [27155](#), 27155
- __fp_add_significand_no_carry_-
o:wwwNN . 1162, 27142, [27145](#), 27145
- __fp_add_significand_o:NnnwnnnnN
1161, 1162, 27110, 27118, [27123](#), 27123
- __fp_add_significand_pack:NNNNNNN
..... [27123](#), 27127, 27130
- __fp_add_significand_test_o:N ..
..... [27123](#), 27125, 27137
- __fp_add_zeros_o:Nww
..... 27047, [27059](#), 27059
- __fp_and_return:wNw
..... [26944](#), 26950, 26957, 26970
- __fp_array_bounds:NNnTF
..... [31159](#), 31159, 31190, 31260
- __fp_array_bounds_error:NNn ...
..... [31159](#), 31162, 31166, 31173
- __fp_array_count:n
[24347](#), 24347, 24944, 24980, 24987,
26691, 26692, 27849, 29989, 30329
- __fp_array_gset:NNNww
..... 31178, 31181, 31188
- __fp_array_gset:w 31178, 31194, 31205
- __fp_array_gset_normal:w
..... 31178, 31209, 31215
- __fp_array_gset_recover:Nw
..... [31178](#), 31195, 31200
- __fp_array_gset_special:nnNNN ..
..... [31178](#),
31208, 31210, 31211, 31223, 31235
- __fp_array_gzero:N 1295
- __fp_array_if_all_fp:nTF
..... [24359](#), 24359, 26335

- __fp_array_if_all_fp_loop:w ...
..... [24359](#), [24361](#), [24364](#), [24367](#)
- \g__fp_array_int
..... [31124](#), [31131](#), [31133](#), [31145](#)
- __fp_array_item:N [31242](#), [31266](#), [31271](#)
- __fp_array_item:NNnN
..... [31242](#), [31261](#), [31264](#)
- __fp_array_item:NwN
..... [31242](#), [31244](#), [31252](#), [31258](#)
- __fp_array_item:w [31242](#), [31274](#), [31276](#)
- __fp_array_item_normal:w
..... [31242](#), [31278](#), [31294](#)
- __fp_array_item_special:w
..... [31242](#), [31273](#), [31282](#)
- \l__fp_array_loop_int
..... [31125](#), [31231](#), [31234](#), [31237](#)
- __fp_array_new:nNNN [31126](#)
- __fp_array_new:nNNNN . [31135](#), [31139](#)
- __fp_array_to_clist:n
.. [25015](#), [30277](#), [30277](#), [30370](#), [30783](#)
- __fp_array_to_clist_loop:Nw ...
..... [30277](#), [30283](#), [30288](#), [30293](#)
- __fp_asec_o:w [29954](#), [29954](#)
- __fp_asin_auxi_o:NnNww
[1255](#), [1257](#), [29919](#), [29922](#), [29922](#), [29981](#)
- __fp_asin_isqrt:wn
..... [29922](#), [29925](#), [29932](#)
- __fp_asin_normal_o:NnwNnnnw ...
..... [29880](#), [29896](#), [29907](#), [29907](#)
- __fp_asin_o:w [29874](#), [29874](#)
- __fp_atan_auxi:ww
..... [1251](#), [29798](#), [29812](#), [29812](#)
- __fp_atan_auxii:w [29812](#), [29813](#), [29814](#)
- __fp_atan_combine_aux:ww
..... [29839](#), [29854](#), [29861](#)
- __fp_atan_combine_o:NwwwwN ...
[1250](#), [1251](#), [29757](#), [29774](#), [29839](#), [29839](#)
- __fp_atan_default:w
[1138](#), [1249](#), [29720](#), [29724](#), [29730](#), [29732](#)
- __fp_atan_div:wnwnw
..... [1251](#), [29785](#), [29787](#), [29787](#)
- __fp_atan_inf_o:NNNw
..... [1249](#), [29745](#), [29746](#),
[29747](#), [29755](#), [29755](#), [29892](#), [29965](#)
- __fp_atan_near:wwn
..... [29787](#), [29794](#), [29800](#)
- __fp_atan_near_aux:wwn
..... [29787](#), [29805](#), [29807](#)
- __fp_atan_normal_o:NNnwNw ...
..... [1249](#), [29749](#), [29765](#), [29765](#)
- __fp_atan_o:Nw
..... [29105](#), [29107](#), [29720](#), [29720](#)
- __fp_atan_Taylor_break:w
..... [29823](#), [29826](#), [29836](#)
- __fp_atan_Taylor_loop:ww
... [1252](#), [29818](#), [29823](#), [29823](#), [29831](#)
- __fp_atan_test_o:NwwNwN
... [1255](#), [29768](#), [29772](#), [29772](#), [29929](#)
- __fp_atanii_o:Nww
..... [29724](#), [29733](#), [29733](#), [29754](#)
- __fp_basics_pack_high:NNNNw ...
..... [1162](#),
[1179](#), [24458](#), [24460](#), [27148](#), [27305](#),
[27410](#), [27423](#), [27568](#), [27761](#), [28304](#)
- __fp_basics_pack_high_carry:w ...
..... [1074](#), [24458](#), [24463](#), [24467](#)
- __fp_basics_pack_low:NNNNw ...
..... [1169](#), [1179](#), [24458](#),
[24458](#), [27150](#), [27307](#), [27412](#), [27425](#),
[27570](#), [27710](#), [27712](#), [27763](#), [28306](#)
- __fp_basics_pack_weird_high:NNNNNNNw
..... [24469](#), [24477](#), [27159](#), [27579](#)
- __fp_basics_pack_weird_low:NNNNw
..... [24469](#), [24469](#), [27161](#), [27581](#)
- __fp_bcmp:ww [26629](#), [26657](#)
- \c__fp_big_leading_shift_int ...
.. [24444](#), [27640](#), [27984](#), [27995](#), [28005](#)
- \c__fp_big_middle_shift_int
.... [24444](#), [27643](#), [27646](#), [27649](#),
[27652](#), [27655](#), [27658](#), [27662](#), [27986](#),
[27997](#), [28007](#), [28017](#), [28020](#), [28023](#)
- \c__fp_big_trailing_shift_int ...
..... [24444](#), [27666](#), [28030](#)
- \c__fp_Bigg_leading_shift_int ...
..... [24449](#), [27487](#), [27505](#)
- \c__fp_Bigg_middle_shift_int ...
.. [24449](#), [27490](#), [27493](#), [27508](#), [27511](#)
- \c__fp_Bigg_trailing_shift_int ..
..... [24449](#), [27496](#), [27514](#)
- __fp_binary_rev_type_o:Nww
..... [25975](#), [25988](#), [27834](#), [27839](#)
- __fp_binary_type_o:Nww
..... [25975](#), [25975](#), [27831](#), [27850](#)
- \c__fp_block_int [24249](#), [28256](#)
- __fp_case_return:nw [1077](#),
[24526](#), [24526](#), [24557](#), [24560](#), [24565](#),
[25076](#), [28537](#), [29036](#), [29745](#), [29746](#),
[29747](#), [30044](#), [30098](#), [30172](#), [30174](#),
[30175](#), [30241](#), [31208](#), [31210](#), [31211](#)
- __fp_case_return_i_o:ww
..... [24533](#), [24533](#),
[27050](#), [27064](#), [27073](#), [27350](#), [29736](#)
- __fp_case_return_ii_o:ww [24533](#),
[24536](#), [27351](#), [28823](#), [28841](#), [29737](#)
- __fp_case_return_o:Nw
..... [1077](#), [1078](#), [24527](#),
[24527](#), [27782](#), [28572](#), [28577](#), [28580](#),

- 28774, 28779, 28802, 28805, 28808,
- 29012, 29126, 29156, 29948, 29950
- __fp_case_return_o:Nww
 [24531](#), [24531](#), [27352](#), [27353](#),
 [27356](#), [27357](#), [28825](#), [28834](#), [28837](#)
- __fp_case_return_same_o:w
 [1077](#), [1078](#),
 [24529](#), [24529](#), [27591](#), [27595](#), [27783](#),
 [27795](#), [27798](#), [28346](#), [28584](#), [28799](#),
 [29016](#), [29019](#), [29111](#), [29119](#), [29134](#),
 [29149](#), [29164](#), [29171](#), [29179](#), [29194](#),
 [29877](#), [29885](#), [29903](#), [29949](#), [29966](#)
- __fp_case_use:nw ... [1077](#), [24525](#),
 [24525](#), [27075](#), [27348](#), [27349](#), [27354](#),
 [27355](#), [27440](#), [27443](#), [27593](#), [27779](#),
 [28339](#), [28342](#), [28810](#), [29022](#), [29112](#),
 [29117](#), [29127](#), [29132](#), [29142](#), [29147](#),
 [29157](#), [29162](#), [29172](#), [29177](#), [29187](#),
 [29192](#), [29879](#), [29882](#), [29892](#), [29894](#),
 [29900](#), [29944](#), [29946](#), [29957](#), [29960](#),
 [29965](#), [30047](#), [30054](#), [30101](#), [30108](#)
- __fp_change_func_type:NNN
 [24387](#), [24387](#), [25764](#), [27826](#), [30029](#),
 [30083](#), [30160](#), [30206](#), [30221](#), [31192](#)
- __fp_change_func_type_aux:w ...
 [24387](#), [24396](#), [24403](#)
- __fp_change_func_type_chk:NNN ..
 [24387](#), [24393](#), [24404](#)
- __fp_chk:w [1063](#)–
 [1065](#), [1120](#), [1159](#), [1161](#), [1163](#), [1169](#),
 [1172](#), [1281](#), [24231](#), [24232](#), [24233](#),
 [24244](#), [24245](#), [24246](#), [24247](#), [24248](#),
 [24258](#), [24263](#), [24265](#), [24266](#), [24294](#),
 [24297](#), [24299](#), [24309](#), [24322](#), [24341](#),
 [24538](#), [24554](#), [24720](#), [24725](#), [24963](#),
 [25019](#), [25028](#), [25030](#), [25862](#), [26504](#),
 [26522](#), [26539](#), [26544](#), [26545](#), [26658](#),
 [26659](#), [26813](#), [26829](#), [26835](#), [26901](#),
 [26902](#), [26905](#), [26916](#), [26917](#), [26925](#),
 [26926](#), [26935](#), [26947](#), [26950](#), [26954](#),
 [26957](#), [27034](#), [27054](#), [27055](#), [27057](#),
 [27058](#), [27059](#), [27067](#), [27070](#), [27081](#),
 [27082](#), [27084](#), [27093](#), [27170](#), [27326](#),
 [27360](#), [27361](#), [27364](#), [27365](#), [27448](#),
 [27449](#), [27589](#), [27597](#), [27599](#), [27776](#),
 [27785](#), [27787](#), [27792](#), [27800](#), [27802](#),
 [27804](#), [27808](#), [28336](#), [28348](#), [28350](#),
 [28569](#), [28586](#), [28588](#), [28771](#), [28790](#),
 [28792](#), [28793](#), [28796](#), [28813](#), [28816](#),
 [28819](#), [28843](#), [28844](#), [28846](#), [28863](#),
 [28953](#), [28966](#), [28968](#), [28972](#), [28976](#),
 [29009](#), [29025](#), [29108](#), [29121](#), [29123](#),
 [29136](#), [29138](#), [29151](#), [29153](#), [29166](#),
 [29168](#), [29181](#), [29183](#), [29196](#), [29206](#),
 [29734](#), [29750](#), [29751](#), [29755](#), [29766](#),
 [29874](#), [29887](#), [29889](#), [29905](#), [29908](#),
 [29918](#), [29941](#), [29952](#), [29954](#), [29968](#),
 [29970](#), [29975](#), [30040](#), [30061](#), [30064](#),
 [30094](#), [30115](#), [30118](#), [30168](#), [30184](#),
 [30187](#), [30262](#), [30263](#), [30347](#), [30349](#),
 [30381](#), [31205](#), [31213](#), [31216](#), [31296](#)
- __fp_clear_function:n
 [31110](#), [31111](#), [31112](#)
- __fp_clear_variable:n
 [30865](#), [30867](#), [30869](#)
- __fp_clear_variable_aux:n
 ... [1287](#), [30865](#), [30873](#), [30875](#), [31045](#)
- __fp_compare:wNNNNw [26225](#)
- __fp_compare_aux:wn
 [26613](#), [26616](#), [26624](#)
- __fp_compare_back:ww [1272](#),
 [26629](#), [26629](#), [26645](#), [26915](#), [30365](#)
- __fp_compare_back_any:ww
 [1147–1149](#),
 [26300](#), [26626](#), [26629](#), [26640](#), [26708](#)
- __fp_compare_back_tuple:ww
 [26685](#), [26685](#)
- __fp_compare_nan:w
 ... [1148](#), [26629](#), [26662](#), [26663](#), [26684](#)
- __fp_compare_npos:nwnw
 [1146](#), [1148](#), [1150](#),
 [26668](#), [26714](#), [26714](#), [27173](#), [28101](#)
- __fp_compare_return:w
 [26597](#), [26599](#), [26602](#)
- __fp_compare_significand:nnnnnnnn
 [26714](#), [26717](#), [26722](#)
- __fp_cos_o:w [29123](#), [29123](#)
- __fp_cot_o:w [1234](#), [29183](#), [29183](#)
- __fp_cot_zero_o:Nnw
 [1233](#), [1234](#), [29141](#), [29183](#), [29186](#), [29198](#)
- __fp_csc_o:w [29138](#), [29138](#)
- __fp_decimate:nNnnnn
 .. [1075](#), [1078](#), [1227](#), [24479](#), [24479](#),
 [24545](#), [24572](#), [25032](#), [27109](#), [27117](#),
 [27199](#), [28615](#), [28619](#), [28991](#), [30124](#)
- __fp_decimate_:Nnnnn . [24491](#), [24491](#)
- __fp_decimate_auxi:Nnnnn [1076](#), [24495](#)
- __fp_decimate_auxii:Nnnnn ... [24495](#)
- __fp_decimate_auxiii:Nnnnn .. [24495](#)
- __fp_decimate_auxiv:Nnnnn ... [24495](#)
- __fp_decimate_auxix:Nnnnn ... [24495](#)
- __fp_decimate_auxv:Nnnnn [24495](#)
- __fp_decimate_auxvi:Nnnnn ... [24495](#)
- __fp_decimate_auxvii:Nnnnn .. [24495](#)
- __fp_decimate_auxviii:Nnnnn . [24495](#)
- __fp_decimate_auxx:Nnnnn [24495](#)
- __fp_decimate_auxxi:Nnnnn ... [24495](#)
- __fp_decimate_auxxii:Nnnnn .. [24495](#)

- __fp_decimate_auxxiii:Nnnnn . [24495](#)
- __fp_decimate_auxxiv:Nnnnn . [24495](#)
- __fp_decimate_auxxv:Nnnnn . [24495](#)
- __fp_decimate_auxxvi:Nnnnn . [24495](#)
- __fp_decimate_pack:nnnnnnnnnw .
..... [1076](#), [24502](#), [24521](#), [24521](#)
- __fp_decimate_pack:nnnnnnnw .
..... [24522](#), [24523](#)
- __fp_decimate_tiny:Nnnnn .
..... [24491](#), [24493](#)
- __fp_div_npos_o:Nww .
.... [1171](#), [1172](#), [27437](#), [27447](#), [27447](#)
- __fp_div_significand_calc:wwnnnnnnn .
..... [1175](#), [27465](#),
[27474](#), [27474](#), [27522](#), [28419](#), [28427](#)
- __fp_div_significand_calc_
i:wwnnnnnnn . [27474](#), [27477](#), [27482](#)
- __fp_div_significand_calc_
ii:wwnnnnnnn . [27474](#), [27479](#), [27500](#)
- __fp_div_significand_i_o:wnnw .
.... [1172](#), [1175](#), [27455](#), [27461](#), [27461](#)
- __fp_div_significand_ii:wnn .
..... [1177](#),
[27469](#), [27470](#), [27471](#), [27518](#), [27518](#)
- __fp_div_significand_iii:wwnnnnn .
..... [1177](#), [27472](#), [27526](#), [27526](#)
- __fp_div_significand_iv:wwnnnnnnn .
..... [1178](#), [27529](#), [27534](#), [27534](#)
- __fp_div_significand_large_
o:wwnnnnnwN .
..... [1180](#), [27561](#), [27575](#), [27575](#)
- __fp_div_significand_pack:NNN .
..... [1179](#), [1213](#), [27520](#),
[27555](#), [27555](#), [28406](#), [28425](#), [28434](#)
- __fp_div_significand_small_
o:wwnnnnnwN .
..... [1179](#), [27559](#), [27565](#), [27565](#)
- __fp_div_significand_test_o:w .
..... [1179](#), [27463](#), [27556](#), [27556](#)
- __fp_div_significand_v:NN .
..... [27540](#), [27542](#), [27545](#)
- __fp_div_significand_v:NNw . [27534](#)
- __fp_div_significand_vi:Nw .
..... [1178](#), [27534](#), [27538](#), [27546](#)
- __fp_division_by_zero_o:Nnw .
..... [1082](#), [24680](#), [24732](#),
[24735](#), [27780](#), [28343](#), [29202](#), [29203](#)
- __fp_division_by_zero_o:NNww .
..... [1082](#), [24689](#),
[24732](#), [24736](#), [27441](#), [27444](#), [28812](#)
- \c__fp_empty_tuple_fp .
..... [24342](#), [25172](#), [25823](#), [25833](#)
- __fp_ep_compare:www .
..... [28095](#), [28095](#), [29781](#)
- __fp_ep_compare_aux:www .
..... [28095](#), [28096](#), [28097](#)
- __fp_ep_div:www .
..... [1247](#), [28127](#), [28127](#), [28241](#),
[29710](#), [29797](#), [29801](#), [29810](#), [29978](#)
- __fp_ep_div_eps_pack:NNNNw .
..... [28158](#), [28162](#), [28164](#), [28167](#)
- __fp_ep_div_epsilon:wnNNNNn . [1202](#)
- __fp_ep_div_epsilon:wnNNNNNn .
..... [28155](#), [28158](#), [28158](#)
- __fp_ep_div_epsilonii:wnNNNNNn .
..... [28158](#), [28160](#), [28169](#)
- __fp_ep_div_esti:www .
..... [1202](#), [28133](#), [28136](#), [28136](#)
- __fp_ep_div_estii:wwnnwn .
..... [28136](#), [28138](#), [28144](#)
- __fp_ep_div_estiii:NNNNwww .
..... [28136](#), [28147](#), [28152](#)
- __fp_ep_inv_to_float_o:wN . [1235](#)
- __fp_ep_inv_to_float_o:wwN . [1245](#),
[28237](#), [28239](#), [28246](#), [29145](#), [29160](#)
- __fp_ep_isqrt:wn . [28182](#), [28182](#), [29939](#)
- __fp_ep_isqrt_aux:wn . [28182](#)
- __fp_ep_isqrt_auxi:wn . [28185](#), [28187](#)
- __fp_ep_isqrt_auxii:wwnnwn .
..... [28182](#), [28189](#), [28195](#)
- __fp_ep_isqrt_epsilon:wN .
..... [1205](#), [28220](#), [28223](#), [28223](#)
- __fp_ep_isqrt_epsilonii:wwN .
.. [28223](#), [28226](#), [28227](#), [28228](#), [28230](#)
- __fp_ep_isqrt_esti:wwnnwn .
..... [28197](#), [28200](#), [28200](#), [28205](#)
- __fp_ep_isqrt_estii:wwnnwn .
..... [28200](#), [28203](#), [28210](#)
- __fp_ep_isqrt_estiii:NNNNwww .
..... [28200](#), [28212](#), [28216](#)
- __fp_ep_mul:www .
..... [1229](#), [28112](#), [28112](#), [29052](#),
[29065](#), [29657](#), [29692](#), [29926](#), [29937](#)
- __fp_ep_mul_raw:wwwN .
.. [28112](#), [28118](#), [28122](#), [29224](#), [29590](#)
- __fp_ep_to_ep:wwN .
..... [28061](#), [28061](#), [28114](#),
[28117](#), [28129](#), [28132](#), [28184](#), [29927](#)
- __fp_ep_to_ep_end:www .
..... [28061](#), [28075](#), [28079](#)
- __fp_ep_to_ep_loop:N . [1244](#), [28061](#),
[28066](#), [28070](#), [28077](#), [28080](#), [29591](#)
- __fp_ep_to_ep_zero:ww .
..... [28061](#), [28085](#), [28093](#)
- __fp_ep_to_fixed:wn . [28042](#), [28042](#),
[29221](#), [29804](#), [29813](#), [29924](#), [30390](#)
- __fp_ep_to_fixed_auxi:www .
..... [28042](#), [28044](#), [28049](#)

__fp_ep_to_fixed_auxii:nnnnnnnwn
 28042, 28056, 28059
 __fp_ep_to_float_o:wN 1235
 __fp_ep_to_float_o:wwN
 1232, 1245, 28237, 28237, 28243,
 28250, 29076, 29115, 29130, 29716
 __fp_error:nnnn
 24647, 24655, 24666, 24683,
 24693, 24723, 24746, 24746, 24748,
 25759, 26338, 26353, 26808, 26828,
 26843, 30035, 30089, 30163, 31202
 __fp_error_num_args:nnnn
 24749, 24749, 24755,
 24955, 24957, 24979, 24985, 30328
 __fp_exp_after_?_f:nw
 1071, 1106, 25156
 __fp_exp_after_any_f:Nnw
 24412, 24412, 24418
 __fp_exp_after_any_f:nw .. 1072,
 24412, 24414, 24438, 25158, 25932
 __fp_exp_after_array_f:w
 1072, 24423, 24432, 24437,
 24438, 25813, 26985, 26996, 27006,
 27014, 30656, 30815, 31018, 31107
 __fp_exp_after_expr_mark_f:nw ..
 1106, 25156, 25164
 __fp_exp_after_expr_stop_f:nw ..
 24412, 24422
 __fp_exp_after_f:nw . 1068, 1106,
 24299, 24309, 24417, 25861, 26003
 __fp_exp_after_normal:nNNw
 24302, 24312, 24329, 24329
 __fp_exp_after_normal:Nwwwww ...
 24331, 24339
 __fp_exp_after_o:w
 1068, 24299, 24299,
 24530, 24535, 24537, 25026, 25070,
 25088, 26320, 26934, 26952, 26961,
 26971, 27058, 27806, 28965, 28970
 __fp_exp_after_special:nNNw ...
 1069, 24304, 24314, 24319, 24319
 __fp_exp_after_symbolic_aux:w ..
 30646, 30649, 30662
 __fp_exp_after_symbolic_f:nw ...
 1282, 30646, 30646,
 30677, 30697, 30719, 30842, 31080
 __fp_exp_after_symbolic_loop:N .
 30646,
 30651, 30668, 30673, 30830, 31065
 __fp_exp_after_tuple_f:nw
 24423, 24424, 24425, 26127
 __fp_exp_after_tuple_o:w 24423,
 24423, 26959, 26962, 26966, 26968
 \c__fp_exp_intarray
 28662, 28748, 28755, 28758, 28761
 __fp_exp_intarray:w
 28719, 28732, 28745
 __fp_exp_intarray_aux:w
 28719, 28753, 28756, 28759, 28763
 __fp_exp_large:NwN 1220,
 28719, 28721, 28727, 28740, 28948
 __fp_exp_large_after:wwn
 1220, 28719, 28738, 28765
 __fp_exp_normal_o:w
 28574, 28588, 28588
 __fp_exp_o:w ... 28313, 28569, 28569
 __fp_exp_overflow:NN
 28588, 28601, 28602, 28629
 __fp_exp_pos_large:NnnNwn
 28620, 28719, 28719
 __fp_exp_pos_o:NNwnw
 28591, 28593, 28596
 __fp_exp_pos_o:Nnnwnw 28588
 __fp_exp_Taylor:Nnnwn
 28616, 28635, 28635, 28767
 __fp_exp_Taylor_break:Nnw
 28635, 28649, 28660
 __fp_exp_Taylor_ii:ww . 28641, 28644
 __fp_exp_Taylor_loop:www
 28635, 28645, 28646, 28655
 __fp_expand:n 1266
 __fp_exponent:w 24266, 24266
 __fp_facorial_int_o:n 1228
 __fp_fact_int_o:n 29030, 29033
 __fp_fact_int_o:w 29027
 __fp_fact_loop_o:w
 29045, 29047, 29047, 29058
 \c__fp_fact_max_arg_int 29008, 29035
 __fp_fact_o:w .. 28317, 29009, 29009
 __fp_fact_pos_o:w 29024, 29027, 29027
 __fp_fact_small_o:w .. 29050, 29062
 \c__fp_five_int 24817,
 24841, 24854, 24867, 24874, 24927
 __fp_fixed(calculation):wwn . 1190
 __fp_fixed_add:nnNnnwn
 27933, 27941, 27943
 __fp_fixed_add:Nnnnnwn
 27933, 27933, 27934, 27935
 __fp_fixed_add:wwn .. 1190, 1193,
 27933, 27933, 28180, 28510, 28519,
 28530, 28548, 29809, 29870, 30405
 __fp_fixed_add_after:NNNNwn ...
 27933, 27937, 27952
 __fp_fixed_add_one:wN
 1191, 27864, 27864,
 28172, 28652, 28661, 29936, 30396

__fp_fixed_add_pack:NNNNwn ...
 27933, 27939, 27946, 27950
 __fp_fixed_continue:wn
 27863, 27863, 28115,
 28120, 28130, 28730, 28923, 29259,
 29637, 29928, 29937, 30388, 30400
 __fp_fixed_div_int:wnN
 27902, 27907, 27915, 27927
 __fp_fixed_div_int:wwN ... 1192,
 27902, 27902, 28509, 28651, 29828
 __fp_fixed_div_int_after:Nw ...
 1193, 27902, 27904, 27932
 __fp_fixed_div_int_auxi:wnn ...
 27902, 27908,
 27909, 27910, 27911, 27912, 27922
 __fp_fixed_div_int_auxii:wnn ...
 1193, 27902, 27913, 27930
 __fp_fixed_div_int_pack:Nw
 1193, 27902, 27925, 27931
 __fp_fixed_div_myriad:wn
 27869, 27869, 28177
 __fp_fixed_inv_to_float_o:wN ...
 28245, 28245, 28593, 28858
 __fp_fixed_mul:nnnnnnnw
 27954, 27974, 27976
 __fp_fixed_mul:wnn .. 1190, 1192,
 1194, 1243, 1245, 27954, 27954,
 28124, 28156, 28171, 28173, 28178,
 28232, 28235, 28249, 28511, 28522,
 28562, 28653, 28751, 28768, 28869,
 29597, 29668, 29816, 29850, 29852
 __fp_fixed_mul_add:nnnnwnnnn ...
 1197, 28024, 28026, 28026
 __fp_fixed_mul_add:nnnnwnnwN ...
 1198, 28031, 28037, 28037
 __fp_fixed_mul_add:Nwnnnwnnn ...
 1197,
 27987, 27998, 28009, 28013, 28013
 __fp_fixed_mul_add:wwnn
 1195, 27981, 27981, 30410
 __fp_fixed_mul_after:wnn
 1195, 27871, 27877, 27877, 27881,
 27956, 27983, 27994, 28004, 28886
 __fp_fixed_mul_one_minus_-
 mul:wnn 27981
 __fp_fixed_mul_short:wnn
 1192, 27879, 27879,
 28154, 28175, 28219, 28221, 29863
 __fp_fixed_mul_sub_back:wwnn ...
 1195, 27981, 27991,
 28233, 29618, 29621, 29623, 29625,
 29627, 29629, 29631, 29633, 29635,
 29640, 29643, 29645, 29647, 29649,
 29651, 29653, 29655, 29681, 29684,
 29686, 29688, 29690, 29694, 29697,
 29699, 29701, 29703, 29829, 29837
 __fp_fixed_one_minus_mul:wnn ...
 1195-1197, 28002
 __fp_fixed_sub:wnn
 27933, 27934, 28225,
 28528, 28544, 28556, 29263, 29810,
 29868, 29934, 30398, 30407, 30439
 __fp_fixed_to_float_o:Nw
 28252, 28252, 28537
 __fp_fixed_to_float_o:wN
 1191, 1206, 1253, 28238,
 28252, 28253, 28254, 28557, 28567,
 28591, 28854, 29858, 30338, 30444
 __fp_fixed_to_float_pack:ww ...
 28285, 28295
 __fp_fixed_to_float_rad_o:wN ...
 28247, 28247, 29858
 __fp_fixed_to_float_round_-
 up:wnnnnw 28298, 28302
 __fp_fixed_to_float_zero:w
 28281, 28290
 __fp_fixed_to_loop:N
 28258, 28268, 28272
 __fp_fixed_to_loop_end:w
 28274, 28278
 __fp_from_dim:wNNnnnnnn
 30231, 30254, 30257
 __fp_from_dim:wnnnnwNn 30258, 30259
 __fp_from_dim:wnnnnwNw 30231
 __fp_from_dim:wNw 30231, 30243, 30252
 __fp_from_dim_test:ww 1265, 25248,
 25285, 25880, 30231, 30233, 30238
 __fp_func_to_name:N
 24592, 24592, 25759, 25768
 __fp_func_to_name_aux:w
 24592, 24595, 24598
 __fp_function_arg_few:w
 31089, 31092, 31104
 __fp_function_arg_get:w
 31089, 31095, 31105
 \l__fp_function_arg_int
 31024, 31041, 31044, 31047, 31055
 __fp_function_arg_o:w ... 31054,
 31059, 31063, 31089, 31089, 31099
 __fp_function_o:w 1280,
 30756, 30989, 31006, 31006, 31013
 __fp_function_set_parsing:Nn ...
 30976, 30979, 30979, 31038, 31118
 __fp_function_set_parsing_-
 aux:NNn 30979, 30981, 30984
 \c__fp_half_prec_int
 24249, 25490, 25522
 __fp_id_if_invalid:n 30787

__fp_id_if_invalid:nTF
 [1287](#), [30786](#), 30871,
 30886, 30906, 30966, 31032, 31114
 __fp_id_if_invalid_aux:N
 [30786](#), 30795, 30801, 30809
 __fp_if_has_symbolic:nTF
 [30637](#), 30637, 30664
 __fp_if_has_symbolic_aux:w
 [30637](#), 30639, 30644
 __fp_if_type_fp:NTwFw
 [1070](#), [1138](#), 24279,
 [24358](#), 24358, 24366, 24373, 24389,
 24416, 26347, 26361, 26605, 26642,
 26643, 26800, 26801, 26802, 26976
 __fp_inf_fp:N .. [24262](#), 24264, 24708
 __fp_int:w [24538](#)
 __fp_int:wTF [24538](#), 30349
 __fp_int_eval:w [1073](#), [1088](#),
 [1090](#), [1105](#), [1120](#), [1161](#), [1169](#), [1170](#),
 [1173](#), [1177](#), [1206](#), [24213](#), 24213,
 24276, 24351, 24483, 24486, 24891,
 24895, 24907, 24908, 24944, 25038,
 25042, 25081, 25295, 25300, 25342,
 25431, 25442, 25492, 25523, 25529,
 25530, 25576, 25586, 25588, 25604,
 25606, 25629, 25631, 25794, 26018,
 26060, 26260, 26618, 27097, 27105,
 27126, 27128, 27149, 27151, 27160,
 27162, 27194, 27200, 27210, 27212,
 27287, 27289, 27306, 27308, 27312,
 27328, 27369, 27377, 27379, 27381,
 27383, 27386, 27389, 27391, 27411,
 27413, 27424, 27426, 27453, 27456,
 27464, 27466, 27487, 27490, 27493,
 27496, 27505, 27508, 27511, 27514,
 27521, 27523, 27530, 27539, 27541,
 27543, 27549, 27569, 27571, 27580,
 27582, 27603, 27624, 27628, 27640,
 27643, 27646, 27649, 27652, 27655,
 27658, 27661, 27665, 27677, 27681,
 27685, 27688, 27709, 27711, 27713,
 27723, 27762, 27764, 27773, 27867,
 27872, 27874, 27882, 27885, 27888,
 27891, 27894, 27897, 27906, 27918,
 27926, 27928, 27938, 27940, 27947,
 27957, 27959, 27962, 27965, 27968,
 27971, 27984, 27986, 27995, 27997,
 28005, 28007, 28017, 28020, 28023,
 28030, 28045, 28064, 28067, 28125,
 28139, 28141, 28148, 28161, 28163,
 28165, 28190, 28206, 28213, 28214,
 28238, 28256, 28260, 28305, 28307,
 28360, 28371, 28390, 28392, 28394,
 28407, 28420, 28426, 28428, 28435,
 28453, 28454, 28455, 28456, 28457,
 28458, 28465, 28467, 28469, 28471,
 28473, 28478, 28480, 28482, 28484,
 28486, 28488, 28514, 28523, 28607,
 28656, 28733, 28741, 28749, 28755,
 28758, 28866, 28887, 28889, 28892,
 28895, 28898, 28901, 28917, 28943,
 28958, 28974, 29044, 29054, 29059,
 29211, 29243, 29252, 29484, 29498,
 29501, 29504, 29507, 29510, 29513,
 29516, 29519, 29522, 29538, 29548,
 29557, 29575, 29584, 29591, 29602,
 29612, 29662, 29672, 29707, 29716,
 29759, 29776, 29778, 29790, 29791,
 29832, 29844, 29855, 29913, 30068,
 30191, 30244, 30314, 30337, 30391,
 30443, 30465, 30467, 30469, 30474,
 30493, 30505, 30513, 30520, 30525
 __fp_int_eval_end: [24213](#),
 24214, 24276, 24354, 24474, 24944,
 25052, 25056, 26261, 26618, 27312,
 27347, 27545, 27928, 28067, 28917,
 28974, 29244, 29253, 29602, 29612,
 29672, 29707, 29791, 30472, 30474
 __fp_int_p:w [24538](#)
 __fp_int_to_roman:w [24213](#), 24215,
 24486, 25504, 25536, 28387, 31133
 __fp_invalid_operation:nnw
 [1081](#), [1082](#), 24644,
 [24732](#), 24732, 24744, 30049, 30056,
 30103, 30110, 30210, 30225, 30728
 __fp_invalid_operation_o:nw ...
 [1082](#), 24743, 24743,
 24745, 25768, 27593, 27819, 28339,
 29022, 29031, 29118, 29133, 29148,
 29163, 29178, 29193, 29883, 29901,
 29917, 29945, 29958, 29974, 30598
 __fp_invalid_operation_o:Nww ...
 [1082](#), 24652, [24732](#), 24733, 25973,
 27077, 27354, 27355, 28959, 30621
 __fp_invalid_operation_o:nnw . 27851
 __fp_invalid_operation_tl_o:nn .
 [1082](#),
 24663, [24732](#), 24734, 25013, 30369
 __fp_kind:w [24277](#), 24277, 25006, 26591
 \c__fp_leading_shift_int
 [24439](#), 27872,
 27882, 27957, 28887, 29538, 29575
 __fp_ln_c:NwNw
 [1214](#), [1215](#), 28489, [28525](#), 28525
 __fp_ln_div_after:Nw
 [1213](#), 28386, 28437
 __fp_ln_div_i:w 28408, 28417

- __fp_ln_div_ii:wnn
... [28411](#), [28412](#), [28413](#), [28414](#), [28422](#)
- __fp_ln_div_vi:wnn ... [28415](#), [28431](#)
- __fp_ln_exponent:wn
..... [1216](#), [28362](#), [28534](#), [28534](#)
- __fp_ln_exponent_one:ww [28539](#), [28553](#)
- __fp_ln_exponent_small:NNww ...
..... [28542](#), [28546](#), [28559](#)
- \c__fp_ln_i_fixed_tl [28318](#)
- \c__fp_ln_ii_fixed_tl [28318](#)
- \c__fp_ln_iii_fixed_tl [28318](#)
- \c__fp_ln_iv_fixed_tl [28318](#)
- \c__fp_ln_ix_fixed_tl [28318](#)
- __fp_ln_npos_o:w
... [1208](#), [1209](#), [28348](#), [28350](#), [28350](#)
- __fp_ln_o:w
... [1208](#), [1224](#), [28315](#), [28336](#), [28336](#)
- __fp_ln_significand:NNNNnnN ...
... [1210](#), [28361](#), [28364](#), [28364](#), [28867](#)
- __fp_ln_square_t_after:w
..... [28464](#), [28497](#)
- __fp_ln_square_t_pack:NNNNw ...
... [28466](#), [28468](#), [28470](#), [28472](#), [28495](#)
- __fp_ln_t_large:NNw
..... [1213](#), [28442](#), [28450](#), [28460](#)
- __fp_ln_t_small:Nw ... [28440](#), [28447](#)
- __fp_ln_t_small:w [1214](#)
- __fp_ln_Taylor:wwNw
..... [1214](#), [28498](#), [28499](#), [28499](#)
- __fp_ln_Taylor_break:w [28507](#), [28518](#)
- __fp_ln_Taylor_loop:www
..... [28501](#), [28504](#), [28513](#)
- __fp_ln_twice_t_after:w [28477](#), [28493](#)
- __fp_ln_twice_t_pack:Nw . [28479](#),
[28481](#), [28483](#), [28485](#), [28487](#), [28492](#)
- \c__fp_ln_vi_fixed_tl [28318](#)
- \c__fp_ln_vii_fixed_tl [28318](#)
- \c__fp_ln_viii_fixed_tl [28318](#)
- \c__fp_ln_x_fixed_tl
..... [28318](#), [28556](#), [28563](#)
- __fp_ln_x_ii:wnnnn
..... [28366](#), [28384](#), [28384](#)
- __fp_ln_x_iii:NNNNNw . [28393](#), [28397](#)
- __fp_ln_x_iii_var:NNNNw
..... [28391](#), [28399](#)
- __fp_ln_x_iv:wnnnnnnnn
..... [1212](#), [28389](#), [28404](#)
- __fp_logb_aux_o:w [27776](#), [27781](#), [27787](#)
- __fp_logb_o:w .. [27023](#), [27776](#), [27776](#)
- \c__fp_max_exp_exponent_int
..... [24255](#), [28599](#)
- \c__fp_max_exponent_int .. [24253](#),
[24259](#), [24287](#), [28084](#), [28292](#), [28922](#)
- \c__fp_middle_shift_int
..... [24439](#), [27885](#),
[27888](#), [27891](#), [27894](#), [27959](#), [27962](#),
[27965](#), [27968](#), [28889](#), [28892](#), [28895](#),
[28898](#), [29541](#), [29548](#), [29578](#), [29584](#)
- __fp_minmax_aux_o:Nw
..... [26887](#), [26891](#), [26893](#)
- __fp_minmax_auxi:ww
..... [26909](#), [26921](#), [26928](#), [26928](#)
- __fp_minmax_auxii:ww
..... [26911](#), [26919](#), [26928](#), [26931](#)
- __fp_minmax_break_o:w
..... [26902](#), [26933](#), [26933](#)
- __fp_minmax_loop:Nww [1154](#),
[26896](#), [26898](#), [26904](#), [26904](#), [26924](#)
- __fp_minmax_o:Nw
... [1146](#), [26584](#), [26586](#), [26887](#), [26887](#)
- \c__fp_minus_min_exponent_int ...
..... [24253](#), [24288](#)
- __fp_misused:n
..... [24229](#), [24229](#), [24233](#), [24344](#)
- __fp_mul_cases_o:NnNww
..... [1171](#), [27319](#), [27325](#), [27434](#)
- __fp_mul_cases_o:nNnww [27325](#)
- __fp_mul_npos_o:Nww . [1168](#), [1169](#),
[1171](#), [1265](#), [27322](#), [27363](#), [27363](#), [30261](#)
- __fp_mul_significand_drop:NNNNw
..... [1169](#), [27373](#),
[27382](#), [27385](#), [27388](#), [27390](#), [27394](#)
- __fp_mul_significand_keep:NNNNw
..... [27373](#), [27378](#), [27380](#), [27396](#)
- __fp_mul_significand_large_
f:NwwNNNN [27403](#), [27407](#), [27407](#)
- __fp_mul_significand_o:nnnnNnnnn
..... [1169](#), [27371](#), [27373](#), [27373](#)
- __fp_mul_significand_small_
f:NNwwwN [27401](#), [27419](#), [27419](#)
- __fp_mul_significand_test_f:NNN
..... [1170](#), [27375](#), [27398](#), [27398](#)
- \c__fp_myriad_int [24252](#),
[27867](#), [27899](#), [27900](#), [27978](#), [28040](#)
- __fp_neg_sign:N
... [1159](#), [24275](#), [24275](#), [27031](#), [27187](#)
- __fp_new_function:n
..... [30962](#), [30963](#), [30964](#)
- __fp_new_variable:n
..... [30880](#), [30882](#), [30884](#)
- __fp_not_o:w [1146](#), [25787](#), [26935](#), [26935](#)
- \c__fp_one_fixed_tl [27861](#),
[28509](#), [28723](#), [28923](#), [28950](#), [29761](#),
[29828](#), [29934](#), [30388](#), [30398](#), [30439](#)
- __fp_overflow:w [1068](#), [1082](#), [1084](#),
[24290](#), [24732](#), [24737](#), [28601](#), [29038](#)

- \c__fp_overflowing_fp [24256](#), [30050](#), [30104](#)
- __fp_pack:NNNNNw [24439](#), [24442](#), [27873](#),
[27884](#), [27887](#), [27890](#), [27893](#), [27896](#),
[27958](#), [27961](#), [27964](#), [27967](#), [27970](#),
[28888](#), [28891](#), [28894](#), [28897](#), [28900](#)
- __fp_pack_big:NNNNNNw [24444](#), [24447](#),
[27642](#), [27645](#), [27648](#), [27651](#), [27654](#),
[27657](#), [27660](#), [27664](#), [27985](#), [27996](#),
[28006](#), [28016](#), [28019](#), [28022](#), [28029](#)
- __fp_pack_Bigg:NNNNNNw [24449](#), [24452](#), [27489](#),
[27492](#), [27495](#), [27507](#), [27510](#), [27513](#)
- __fp_pack_eight:wNNNNNNNN [1074](#), [1166](#), [24456](#), [24456](#),
[27297](#), [27613](#), [28052](#), [29230](#), [29231](#)
- __fp_pack_twice_four:wNNNNNNNN [1074](#), [24454](#), [24454](#), [25063](#), [25064](#),
[27238](#), [27239](#), [28053](#), [28054](#), [28055](#),
[28087](#), [28088](#), [28089](#), [28283](#), [28284](#),
[28638](#), [28639](#), [28640](#), [29232](#), [29233](#),
[29527](#), [29528](#), [29529](#), [29530](#), [30254](#)
- __fp_parse:n .. [1096](#), [1108](#), [1120](#),
[1128](#), [1141](#), [1142](#), [1151](#), [1266](#), [1295](#),
[25094](#), [25245](#), [25904](#), [25904](#), [26458](#),
[26460](#), [26462](#), [26485](#), [26591](#), [26600](#),
[26617](#), [26627](#), [26795](#), [26851](#), [27789](#),
[30025](#), [30079](#), [30157](#), [30202](#), [30217](#),
[30270](#), [30272](#), [30274](#), [30276](#), [31185](#)
- __fp_parse_after:ww [25904](#), [25909](#), [25917](#), [25924](#)
- __fp_parse_apply_binary:NwNwN .. [1100](#),
[1101](#), [1104](#), [1132](#), [25946](#), [25946](#), [26137](#)
- __fp_parse_apply_binary_chk:NN .. [25946](#), [25951](#), [25963](#), [25977](#), [25990](#)
- __fp_parse_apply_binary_-
error:NNN [25946](#), [25966](#), [25970](#)
- __fp_parse_apply_comma:NwNwN ... [1132](#), [26096](#), [26107](#), [26122](#)
- __fp_parse_apply_compare:NwNNNNNwN [26284](#), [26293](#)
- __fp_parse_apply_compare_-
aux:NNwN [26305](#), [26308](#), [26313](#)
- __fp_parse_apply_function:NNNwN [1123](#), [25736](#), [25736](#), [25897](#)
- __fp_parse_apply_unary:NNNwN ... [25741](#), [25741](#), [25773](#), [25888](#)
- __fp_parse_apply_unary_chk:nNNNNw [25752](#), [25753](#), [25756](#)
- __fp_parse_apply_unary_chk:nNNNw [25741](#)
- __fp_parse_apply_unary_chk:NwNw [25741](#), [25743](#), [25748](#)
- __fp_parse_apply_unary_error:NNw [25741](#), [25764](#), [25767](#), [27827](#)
- __fp_parse_apply_unary_type:NNN [25741](#), [25744](#), [25762](#)
- __fp_parse_caseless_inf:N [25854](#), [25854](#)
- __fp_parse_caseless_infinity:N . [25854](#), [25855](#)
- __fp_parse_caseless_nan:N [25854](#), [25856](#)
- __fp_parse_compare:NNNNNNN [26225](#), [26226](#), [26228](#),
[26230](#), [26233](#), [26246](#), [26254](#), [26315](#)
- __fp_parse_compare_auxi:NNNNNNN [26225](#), [26249](#), [26257](#), [26271](#)
- __fp_parse_compare_auxii:NNNNN . [26225](#),
[26262](#), [26263](#), [26264](#), [26265](#), [26269](#)
- __fp_parse_compare_end:NNNNw ... [26225](#), [26266](#), [26280](#)
- __fp_parse_continue:NwN [1100](#), [1101](#),
[1128](#), [25935](#), [25938](#), [25945](#), [25948](#),
[26124](#), [26323](#), [26993](#), [27003](#), [27011](#)
- __fp_parse_continue_compare:NNwNN [26316](#), [26331](#)
- __fp_parse_digits:N [25112](#), [25130](#), [25131](#)
- __fp_parse_digits_i:N . [25112](#), [25129](#)
- __fp_parse_digits_ii:N [25112](#), [25128](#)
- __fp_parse_digits_iii:N [25112](#), [25127](#)
- __fp_parse_digits_iv:N [25112](#), [25126](#)
- __fp_parse_digits_v:N . [25112](#), [25125](#)
- __fp_parse_digits_vi:N [25112](#), [25124](#), [25447](#), [25496](#)
- __fp_parse_digits_vii:N [1113](#), [25112](#), [25434](#), [25485](#)
- __fp_parse_excl_error: [26225](#), [26241](#), [26250](#)
- __fp_parse_expand:w [1104](#), [1105](#), [25109](#), [25109](#), [25111](#),
[25121](#), [25161](#), [25221](#), [25265](#), [25274](#),
[25277](#), [25281](#), [25318](#), [25352](#), [25390](#),
[25392](#), [25411](#), [25413](#), [25435](#), [25452](#),
[25465](#), [25486](#), [25516](#), [25544](#), [25560](#),
[25571](#), [25594](#), [25623](#), [25633](#), [25640](#),
[25654](#), [25670](#), [25690](#), [25701](#), [25783](#),
[25806](#), [25818](#), [25893](#), [25902](#), [25912](#),
[25927](#), [26045](#), [26091](#), [26115](#), [26141](#),
[26189](#), [26209](#), [26278](#), [26291](#), [26989](#)
- __fp_parse_exponent:N [1118](#), [25220](#),
[25426](#), [25576](#), [25643](#), [25645](#), [25645](#)

__fp_parse_exponent:Nw
 25450, 25463, 25513,
 25541, 25592, 25621, 25640, 25640
 __fp_parse_exponent_aux:NN
 25645, 25648, 25656
 __fp_parse_exponent_body:N
 25672, 25676, 25676
 __fp_parse_exponent_digits:N
 25680, 25692, 25692, 25696
 __fp_parse_exponent_keep:N .. 25703
 __fp_parse_exponent_keep:NTF
 25683, 25703
 __fp_parse_exponent_sign:N
 25662, 25666, 25666, 25669
 __fp_parse_function:NNN
 24804, 24806, 24808, 24811, 25886,
 25895, 26584, 26586, 29101, 29103,
 29105, 29107, 30299, 30301, 30988
 __fp_parse_function_all_fp_
 o:nnw ... 24937, 26333, 26333, 26889
 __fp_parse_function_one_two:nnw
 1249,
 26345, 26345, 29722, 29728, 30342
 __fp_parse_function_one_two_
 aux:nnw 26345, 26349, 26359
 __fp_parse_function_one_two_
 auxii:nnw 26345, 26371, 26373
 __fp_parse_function_one_two_
 error_o:w
 .. 26345, 26348, 26351, 26368, 26376
 __fp_parse_infix:NN
 ... 1106, 1110, 1126, 1131, 1132,
 25160, 25330, 25369, 25846, 25861,
 25883, 26003, 26006, 26089, 30842
 __fp_parse_infix!:N 26225
 __fp_parse_infix_&:Nw 26182
 __fp_parse_infix(:N 26165
 __fp_parse_infix):N 26079
 __fp_parse_infix*:N 26167
 __fp_parse_infix+:N
 1104, 25109, 26131
 __fp_parse_infix_,:N 26096
 __fp_parse_infix -:N 26131
 __fp_parse_infix/:N 26131
 __fp_parse_infix::N . 26199, 26974
 __fp_parse_infix<:N 26225
 __fp_parse_infix=:N 26225
 __fp_parse_infix>:N 26225
 __fp_parse_infix?:N 26199
 __fp_parse_infix⟨operation₂⟩:N 1104
 __fp_parse_infix^:N 26131
 __fp_parse_infix_after_operand:NwN
 1110,
 25213, 25291, 25790, 26001, 26001
 __fp_parse_infix_after_paren:NN
 25815, 25841, 26048, 26048
 __fp_parse_infix_and:N 26131, 26198
 __fp_parse_infix_check:NNN
 26024, 26034, 26066
 __fp_parse_infix_comma:w
 1132, 26096, 26111, 26120
 __fp_parse_infix_end:N
 1128, 1132, 25913,
 25918, 25928, 26077, 26077, 26078
 __fp_parse_infix_juxt:N
 1131, 26014, 26022, 26131
 __fp_parse_infix_mark:NNN
 26011, 26053, 26076, 26076
 __fp_parse_infix_mul:N
 1131, 1134, 26039,
 26056, 26064, 26131, 26166, 26175
 __fp_parse_infix_or:N . 26131, 26197
 __fp_parse_infix_|:Nw 26182
 __fp_parse_large:N
 1112, 25397, 25481, 25481
 __fp_parse_large_leading:wwNN ..
 1116, 25483, 25488, 25488
 __fp_parse_large_round:NN
 1116, 25524, 25596, 25596
 __fp_parse_large_round_aux:wNN .
 25596, 25605, 25625
 __fp_parse_large_round_test:NN .
 25596, 25609, 25614
 __fp_parse_large_trailing:wwNN .
 1116, 25494, 25518, 25518
 __fp_parse_letters:N
 ... 1110, 25306, 25320, 25335, 25347
 __fp_parse_lparen_after:NwN ...
 25796, 25798, 25808
 __fp_parse_o:n
 ... 1096, 25904, 25919, 26793, 26794
 __fp_parse_one:Nw 1099-
 1104, 1111, 1126, 1128, 25109,
 25132, 25132, 25374, 25735, 25941
 __fp_parse_one_digit:NN
 1124, 25148, 25289, 25289
 __fp_parse_one_fp:NN
 1106, 25140, 25156, 25156
 __fp_parse_one_other:NN
 25151, 25297, 25297
 __fp_parse_one_register:NN
 25143, 25211, 25211
 __fp_parse_one_register_aux:Nw .
 25211, 25217, 25223
 __fp_parse_one_register_
 auxii:wwNw ... 25211, 25228, 25237
 __fp_parse_one_register_dim:ww .
 25211, 25231, 25243, 25246

__fp_parse_one_register_int:www
 25211, 25233, 25244
 __fp_parse_one_register_-
 math:NNw 25252, 25258, 25261, 25264
 __fp_parse_one_register_mu:www .
 25211, 25232, 25241
 __fp_parse_one_register_-
 special:N 25216, 25252, 25252
 __fp_parse_one_register_wd:Nw ..
 25252, 25280, 25283
 __fp_parse_one_register_wd:w ...
 .. 25252, 25254, 25255, 25256, 25276
 __fp_parse_operand:Nw 1099-
 1102, 1104, 1128, 1132, 25109,
 25779, 25781, 25802, 25804, 25893,
 25902, 25911, 25926, 25935, 25935,
 26114, 26140, 26208, 26291, 26988
 __fp_parse_pack_carry:w
 1115, 25467, 25476, 25479
 __fp_parse_pack_leading:NNNNNww
 25430, 25467, 25473, 25491
 __fp_parse_pack_trailing:NNNNNNww
 25440,
 25467, 25467, 25510, 25521, 25528
 __fp_parse_prefix:NNN
 25309, 25354, 25354
 __fp_parse_prefix_!:Nw 25769
 __fp_parse_prefix_(:Nw 25796
 __fp_parse_prefix_):Nw 25828
 __fp_parse_prefix_+:Nw 25735
 __fp_parse_prefix_-:Nw 25769
 __fp_parse_prefix_:Nw 25788
 __fp_parse_prefix_unknown:NNN ..
 25354, 25357, 25362
 __fp_parse_return_sep:w
 25110, 25110, 25119, 25350,
 25558, 25569, 25652, 25684, 25699
 __fp_parse_round:Nw .. 24809, 24815
 __fp_parse_round_after:wN 1118,
 25573, 25573, 25578, 25587, 25628
 __fp_parse_round_loop:N
 1118, 1119, 25546,
 25546, 25551, 25589, 25607, 25632
 __fp_parse_round_up:N
 25546, 25554, 25562, 25566
 __fp_parse_small:N
 1113, 25417, 25428, 25428
 __fp_parse_small_leading:wwNN ..
 ... 1114, 25432, 25437, 25437, 25500
 __fp_parse_small_round:NN
 25459, 25578, 25578, 25617
 __fp_parse_small_trailing:wwNN .
 ... 1114, 25445, 25454, 25454, 25532
 __fp_parse_strim_end:w
 25403, 25409, 25413
 __fp_parse_strim_zeros:N
 1112, 1124,
 25384, 25403, 25403, 25407, 25794
 __fp_parse_trim_end:w
 25377, 25387, 25392
 __fp_parse_trim_zeros:N
 25295, 25377, 25377, 25380
 __fp_parse_unary_function:NNN ..
 25886,
 25886, 27021, 27023, 27025, 27027,
 28313, 28315, 28317, 29089, 29095
 __fp_parse_word:Nw
 1110, 25303, 25320, 25320
 __fp_parse_word_abs:N . 27020, 27020
 __fp_parse_word_acos:N 29081
 __fp_parse_word_acosd:N 29081
 __fp_parse_word_acot:N 29100, 29100
 __fp_parse_word_acotd:N 29100, 29102
 __fp_parse_word_acsc:N 29081
 __fp_parse_word_acscd:N 29081
 __fp_parse_word_asec:N 29081
 __fp_parse_word_asecd:N 29081
 __fp_parse_word_asin:N 29081
 __fp_parse_word_asind:N 29081
 __fp_parse_word_atan:N 29100, 29104
 __fp_parse_word_atand:N 29100, 29106
 __fp_parse_word_bp:N 25857
 __fp_parse_word_cc:N 25857
 __fp_parse_word_ceil:N 24803, 24807
 __fp_parse_word_cm:N 25857
 __fp_parse_word_cos:N 29081
 __fp_parse_word_cosd:N 29081
 __fp_parse_word_cot:N 29081
 __fp_parse_word_cotd:N 29081
 __fp_parse_word_csc:N 29081
 __fp_parse_word_cscd:N 29081
 __fp_parse_word_dd:N 25857
 __fp_parse_word_deg:N 25843
 __fp_parse_word_em:N 25876
 __fp_parse_word_ex:N 25876
 __fp_parse_word_exp:N . 28312, 28312
 __fp_parse_word_fact:N 28312, 28316
 __fp_parse_word_false:N 25843
 __fp_parse_word_floor:N 24803, 24805
 __fp_parse_word_in:N 25857
 __fp_parse_word_inf:N
 25843, 25854, 25855
 __fp_parse_word_ln:N . 28312, 28314
 __fp_parse_word_logb:N 27020, 27022
 __fp_parse_word_max:N . 26583, 26583
 __fp_parse_word_min:N . 26583, 26585
 __fp_parse_word_mm:N 25857

- __fp_parse_word_nan:N . [25843](#), [25856](#)
- __fp_parse_word_nc:N [25857](#)
- __fp_parse_word_nd:N [25857](#)
- __fp_parse_word_pc:N [25857](#)
- __fp_parse_word_pi:N [25843](#)
- __fp_parse_word_pt:N [25857](#)
- __fp_parse_word_rand:N [30298](#), [30298](#)
- __fp_parse_word_randint:N
 [30298](#), [30300](#)
- __fp_parse_word_round:N [24809](#), [24809](#)
- __fp_parse_word_sec:N [29081](#)
- __fp_parse_word_secd:N [29081](#)
- __fp_parse_word_sign:N [27020](#), [27024](#)
- __fp_parse_word_sin:N [29081](#)
- __fp_parse_word_sind:N [29081](#)
- __fp_parse_word_sp:N [25857](#)
- __fp_parse_word_sqrt:N [27020](#), [27026](#)
- __fp_parse_word_tan:N [29081](#)
- __fp_parse_word_tand:N [29081](#)
- __fp_parse_word_true:N [25843](#)
- __fp_parse_word_trunc:N [24803](#), [24803](#)
- __fp_parse_zero:
 . . . [1112](#), [25399](#), [25419](#), [25423](#), [25423](#)
- __fp_pow_B:wwN [28870](#), [28905](#)
- __fp_pow_C_neg:w [28908](#), [28925](#)
- __fp_pow_C_overflow:w
 [28913](#), [28920](#), [28941](#)
- __fp_pow_C_pack:w [28927](#), [28935](#), [28946](#)
- __fp_pow_C_pos:w [28911](#), [28930](#)
- __fp_pow_C_pos_loop:wN
 [28931](#), [28932](#), [28939](#)
- __fp_pow_exponent:Nwnnnnw
 [28876](#), [28879](#), [28884](#)
- __fp_pow_exponent:wnN . [28868](#), [28873](#)
- __fp_pow_neg:www
 [1226](#), [28781](#), [28952](#), [28952](#)
- __fp_pow_neg_aux:wNN
 [1226](#), [28952](#), [28956](#), [28968](#)
- __fp_pow_neg_case:w
 [28955](#), [28976](#), [28976](#)
- __fp_pow_neg_case_aux:nnnn
 [28976](#), [28980](#), [28986](#)
- __fp_pow_neg_case_aux:Nnnw
 [1227](#), [28976](#), [28992](#), [28996](#)
- __fp_pow_normal_o:ww
 [1222](#), [28786](#), [28818](#), [28818](#)
- __fp_pow_npos_aux:NNnw
 [28852](#), [28856](#), [28862](#), [28862](#)
- __fp_pow_npos_o:Nww
 [1223](#), [28829](#), [28846](#), [28846](#)
- __fp_pow_zero_or_inf:ww
 [1222](#), [28788](#), [28795](#), [28795](#)
- \c__fp_prec_and_int . . . [25094](#), [26162](#)
- \c__fp_prec_colon_int
 [25094](#), [26220](#), [26988](#)
- \c__fp_prec_comma_int
 [1125](#), [25094](#), [25168](#),
 [25802](#), [25830](#), [26100](#), [26105](#), [26114](#)
- \c__fp_prec_comp_int
 [25094](#), [26248](#), [26291](#)
- \c__fp_prec_end_int . . [1128](#), [1132](#),
 [25094](#), [25170](#), [25911](#), [25926](#), [26083](#)
- \c__fp_prec_func_int
 . . . [1125](#), [25094](#), [25801](#), [25893](#), [25902](#)
- \c__fp_prec_hat_int . . . [25094](#), [26150](#)
- \c__fp_prec_hatii_int . [25094](#), [26150](#)
- \c__fp_prec_int . . . [16973](#), [16980](#),
 [24249](#), [24483](#), [24545](#), [24572](#), [25032](#),
 [28619](#), [28988](#), [28991](#), [30122](#), [30124](#),
 [30130](#), [30181](#), [30353](#), [30392](#), [30443](#)
- \c__fp_prec_juxt_int . . [25094](#), [26152](#)
- \c__fp_prec_not_int
 [1124](#), [25094](#), [25786](#), [25787](#)
- \c__fp_prec_or_int [25094](#), [26164](#)
- \c__fp_prec_plus_int
 [1099](#), [25094](#), [26158](#), [26160](#)
- \c__fp_prec_quest_int
 [25094](#), [26203](#), [26218](#)
- \c__fp_prec_times_int
 [25094](#), [26154](#), [26156](#)
- \c__fp_prec_tuple_int
 . . . [1125](#), [25094](#), [25169](#), [25804](#), [25832](#)
- __fp_rand_myriads:n
 [1271](#), [1272](#), [30308](#), [30308](#), [30325](#), [30411](#)
- __fp_rand_myriads_get:w
 [30308](#), [30313](#), [30318](#)
- __fp_rand_myriads_loop:w
 [30308](#), [30309](#), [30310](#), [30316](#)
- __fp_rand_o:Nw . [30299](#), [30319](#), [30319](#)
- __fp_rand_o:w . . [30319](#), [30323](#), [30333](#)
- __fp_randinat_wide_aux:w . . . [30481](#)
- __fp_randinat_wide_auxii:w . . [30481](#)
- __fp_randint:n . [30546](#), [30549](#), [30551](#)
- __fp_randint:ww
 . . [30449](#), [30453](#), [30458](#), [30463](#), [30556](#)
- __fp_randint_auxi_o:ww
 [30340](#), [30367](#), [30375](#)
- __fp_randint_auxii:wn
 [30340](#), [30378](#), [30379](#), [30381](#)
- __fp_randint_auxiii_o:ww
 [30340](#), [30379](#), [30403](#)
- __fp_randint_auxiv_o:ww
 [30340](#), [30414](#), [30418](#)
- __fp_randint_auxv_o:w
 [30340](#), [30416](#), [30426](#), [30428](#)
- __fp_randint_badarg:w
 . . . [1272](#), [30340](#), [30347](#), [30363](#), [30364](#)

```

\__fp_randint_default:w .....
    ..... 30340, 30344, 30346
\__fp_randint_o:Nw 30301, 30340, 30340
\__fp_randint_o:w 30340, 30344, 30360
\__fp_randint_split_aux:w .....
    ..... 30481, 30504, 30510
\__fp_randint_split_o:Nw .....
    ..... 1275, 30481,
    30486, 30489, 30492, 30494, 30499
\__fp_randint_wide_aux:w .....
    ..... 1275, 30484, 30515
\__fp_randint_wide_auxii:w .....
    ..... 30519, 30528
\__fp_reverse_args:Nww .....
    ..... 1255, 1256, 24223, 24223,
    29708, 29783, 29897, 29963, 30437
\__fp_round:NNN .....
    ..... 1088, 1090, 1171, 1186,
    24818, 24885, 24888, 27152, 27163,
    27414, 27427, 27572, 27583, 27767
\__fp_round:Nwn .....
    .. 24946, 25002, 25004, 25019, 30229
\__fp_round:Nww .....
    ..... 24947, 24969, 25002, 25002
\__fp_round:Nwww . 24948, 24962, 24962
\__fp_round_aux_o:Nw .....
    ..... 24935, 24939, 24941
\__fp_round_digit:Nw .....
    ..... 1076, 1090, 1169, 1171,
    1186, 24501, 24902, 24902, 27166,
    27314, 27417, 27430, 27586, 27772
\__fp_round_name_from_cs:N 24938,
    24958, 24986, 24991, 24991, 25014
\__fp_round_neg:NNN .....
    ..... 1088, 1091, 1167,
    24913, 24934, 27275, 27290, 27309
\__fp_round_no_arg_o:Nw .....
    ..... 24945, 24952, 24952
\__fp_round_normal:NnnwNNnn .....
    ..... 25002, 25033, 25035
\__fp_round_normal:NNwNnn .....
    ..... 25002, 25037, 25057
\__fp_round_normal:NwNNnw .....
    ..... 25002, 25022, 25030
\__fp_round_normal_end:wwNnn ...
    ..... 25002, 25065, 25068
\__fp_round_o:Nw ..... 24804,
    24806, 24808, 24812, 24935, 24935
\__fp_round_pack:Nw .....
    ..... 25002, 25041, 25055
\__fp_round_return_one: .....
    ..... 1088, 24818, 24824,
    24834, 24842, 24846, 24855, 24859,
    24868, 24875, 24879, 24917, 24928
\__fp_round_s:NNNw ..... 1088,
    1090, 1118, 24886, 24886, 25582, 25600
\__fp_round_special:NwwNnn .....
    ..... 25002, 25060, 25073
\__fp_round_special_aux:Nw .....
    ..... 25002, 25079, 25086
\__fp_round_to_nearest:NNN 1091,
    1092, 24812, 24815, 24818, 24839,
    24885, 24922, 24954, 24965, 30229
\__fp_round_to_nearest_neg:NNN ..
    ..... 24913, 24922, 24934
\__fp_round_to_nearest_ninf:NNN .
    ..... 1092, 24818, 24852, 24933
\__fp_round_to_nearest_ninf-
    neg:NNN ..... 24913, 24923
\__fp_round_to_nearest_pinf:NNN .
    ..... 1092, 24818, 24872, 24924
\__fp_round_to_nearest_pinf-
    neg:NNN ..... 24913, 24932
\__fp_round_to_nearest_zero:NNN .
    ..... 1092, 24818, 24865
\__fp_round_to_nearest_zero-
    neg:NNN ..... 24913, 24925
\__fp_round_to_ninf:NNN .....
    .. 24806, 24818, 24820, 24921, 24995
\__fp_round_to_ninf_neg:NNN ....
    ..... 24913, 24913
\__fp_round_to_pinf:NNN .....
    .. 24808, 24818, 24830, 24913, 24997
\__fp_round_to_pinf_neg:NNN ....
    ..... 24913, 24921
\__fp_round_to_zero:NNN .....
    ..... 24804, 24818, 24829, 24993
\__fp_round_to_zero_neg:NNN ....
    ..... 24913, 24914
\__fp_rrrot:www .. 24225, 24225, 29829
\__fp_sanitize:Nw .... 1161, 1164,
    1169, 1172, 1180, 1228, 1245, 1253,
    1272, 24284, 24284, 24296, 25071,
    25089, 27095, 27192, 27367, 27451,
    27601, 28352, 28605, 28848, 29042,
    29660, 29714, 29842, 30335, 30430
\__fp_sanitize:wN .....
    1109, 1113, 24284, 24296, 25294, 25793
\__fp_sanitize_zero:w .....
    ..... 24284, 24292, 24297
\__fp_sec_o:w ..... 29153, 29153
\__fp_sep: .....
    1063–1065, 1069, 1070, 1072–1075,
    1077, 1078, 1088, 1090, 1096, 1098,
    1105, 1114, 1116–1118, 1120, 1122,
    1123, 1146, 1148, 1150, 1161–1166,
    1169, 1170, 1172, 1175–1180, 1182,

```

1184, 1190–1198, 1206, 1210, 1212–
1216, 1218, 1220, 1222, 1229, 1243–
1245, 1250, 1252, 1265, 1271, 1275,
1280, 1281, 24216, 24216, 24218,
24219, 24220, 24221, 24222, 24223,
24224, 24225, 24226, 24227, 24228,
24232, 24233, 24244, 24245, 24246,
24247, 24248, 24260, 24263, 24265,
24284, 24296, 24297, 24319, 24326,
24329, 24333, 24334, 24335, 24336,
24337, 24340, 24341, 24343, 24344,
24346, 24348, 24349, 24352, 24356,
24361, 24364, 24426, 24434, 24442,
24443, 24447, 24448, 24452, 24453,
24454, 24455, 24456, 24457, 24458,
24459, 24460, 24465, 24467, 24468,
24469, 24475, 24478, 24492, 24494,
24501, 24502, 24526, 24527, 24531,
24534, 24535, 24536, 24538, 24557,
24563, 24569, 24570, 24576, 24586,
24644, 24647, 24652, 24656, 24657,
24680, 24683, 24689, 24694, 24695,
24720, 24725, 24732, 24733, 24735,
24736, 24886, 24900, 24902, 24963,
24976, 25002, 25004, 25006, 25015,
25019, 25028, 25030, 25035, 25053,
25066, 25068, 25071, 25073, 25084,
25086, 25089, 25111, 25124, 25125,
25126, 25127, 25128, 25129, 25130,
25131, 25231, 25232, 25233, 25238,
25239, 25242, 25243, 25244, 25246,
25250, 25283, 25287, 25320, 25425,
25437, 25454, 25468, 25471, 25473,
25477, 25479, 25480, 25488, 25518,
25573, 25575, 25625, 25635, 25642,
25660, 25686, 25743, 25748, 25862,
25882, 25975, 25986, 25988, 25999,
26001, 26004, 26128, 26313, 26320,
26340, 26355, 26359, 26371, 26373,
26378, 26380, 26387, 26389, 26391,
26394, 26399, 26400, 26407, 26408,
26410, 26412, 26416, 26504, 26505,
26506, 26522, 26527, 26532, 26539,
26544, 26545, 26550, 26555, 26559,
26564, 26602, 26624, 26627, 26629,
26638, 26640, 26655, 26658, 26659,
26668, 26685, 26686, 26688, 26689,
26696, 26697, 26703, 26708, 26714,
26798, 26804, 26808, 26809, 26813,
26829, 26833, 26835, 26836, 26901,
26902, 26905, 26916, 26917, 26925,
26926, 26929, 26930, 26931, 26933,
26934, 26935, 26947, 26950, 26954,
26957, 26961, 26962, 26965, 26966,
26967, 26968, 26970, 26971, 27034,
27054, 27057, 27070, 27081, 27093,
27105, 27107, 27115, 27123, 27128,
27135, 27146, 27153, 27156, 27166,
27167, 27170, 27173, 27180, 27182,
27184, 27188, 27190, 27212, 27214,
27215, 27216, 27221, 27223, 27226,
27228, 27234, 27236, 27241, 27243,
27245, 27246, 27247, 27248, 27265,
27267, 27280, 27281, 27283, 27293,
27300, 27303, 27314, 27315, 27326,
27360, 27364, 27365, 27391, 27392,
27394, 27395, 27396, 27397, 27408,
27420, 27448, 27449, 27456, 27458,
27461, 27466, 27467, 27483, 27486,
27497, 27501, 27504, 27515, 27518,
27524, 27526, 27530, 27531, 27535,
27543, 27546, 27553, 27555, 27566,
27573, 27576, 27586, 27587, 27589,
27597, 27599, 27608, 27610, 27615,
27618, 27619, 27620, 27628, 27629,
27631, 27667, 27669, 27673, 27693,
27695, 27697, 27699, 27713, 27715,
27724, 27725, 27727, 27734, 27752,
27754, 27757, 27759, 27773, 27774,
27776, 27785, 27787, 27792, 27800,
27802, 27804, 27812, 27824, 27828,
27830, 27831, 27832, 27834, 27835,
27837, 27839, 27840, 27845, 27846,
27852, 27853, 27862, 27863, 27864,
27867, 27869, 27875, 27877, 27878,
27879, 27900, 27902, 27908, 27909,
27910, 27911, 27912, 27913, 27915,
27918, 27922, 27930, 27931, 27932,
27935, 27943, 27948, 27950, 27951,
27952, 27954, 27976, 27979, 27981,
27988, 27989, 27992, 27999, 28000,
28002, 28010, 28011, 28013, 28026,
28034, 28037, 28045, 28047, 28050,
28056, 28059, 28060, 28061, 28068,
28080, 28083, 28090, 28091, 28093,
28094, 28095, 28096, 28098, 28102,
28112, 28114, 28117, 28122, 28124,
28127, 28129, 28132, 28136, 28140,
28142, 28145, 28149, 28150, 28152,
28154, 28158, 28165, 28167, 28168,
28169, 28171, 28173, 28176, 28180,
28182, 28184, 28195, 28198, 28214,
28217, 28219, 28221, 28223, 28225,
28226, 28227, 28228, 28230, 28232,
28233, 28234, 28235, 28239, 28242,
28247, 28249, 28252, 28253, 28254,
28265, 28266, 28278, 28280, 28286,
28288, 28290, 28292, 28293, 28295,

28300, 28302, 28307, 28319, 28321,
 28323, 28325, 28327, 28329, 28331,
 28333, 28335, 28336, 28348, 28350,
 28382, 28384, 28394, 28397, 28398,
 28399, 28401, 28404, 28408, 28409,
 28417, 28420, 28423, 28428, 28429,
 28432, 28435, 28437, 28448, 28453,
 28454, 28455, 28456, 28457, 28458,
 28462, 28476, 28488, 28492, 28493,
 28494, 28495, 28496, 28497, 28502,
 28504, 28509, 28510, 28511, 28514,
 28516, 28519, 28523, 28525, 28532,
 28534, 28551, 28553, 28556, 28559,
 28564, 28566, 28569, 28586, 28596,
 28635, 28642, 28644, 28645, 28646,
 28651, 28653, 28656, 28657, 28660,
 28661, 28719, 28725, 28727, 28733,
 28735, 28745, 28761, 28763, 28764,
 28765, 28767, 28768, 28771, 28790,
 28792, 28793, 28796, 28813, 28819,
 28843, 28863, 28869, 28870, 28873,
 28882, 28884, 28903, 28905, 28916,
 28918, 28920, 28925, 28930, 28931,
 28932, 28943, 28953, 28955, 28959,
 28966, 28976, 28996, 29009, 29025,
 29027, 29029, 29031, 29045, 29047,
 29055, 29056, 29062, 29074, 29075,
 29108, 29121, 29123, 29136, 29138,
 29151, 29153, 29166, 29168, 29181,
 29183, 29196, 29206, 29218, 29220,
 29221, 29222, 29225, 29228, 29235,
 29240, 29245, 29246, 29248, 29253,
 29255, 29263, 29264, 29481, 29485,
 29486, 29531, 29534, 29543, 29545,
 29550, 29552, 29553, 29557, 29560,
 29579, 29581, 29586, 29588, 29591,
 29592, 29595, 29597, 29610, 29618,
 29619, 29620, 29621, 29622, 29623,
 29624, 29625, 29626, 29627, 29628,
 29629, 29630, 29631, 29632, 29633,
 29634, 29635, 29636, 29640, 29641,
 29642, 29643, 29644, 29645, 29646,
 29647, 29648, 29649, 29650, 29651,
 29652, 29653, 29654, 29655, 29656,
 29657, 29666, 29668, 29679, 29681,
 29682, 29683, 29684, 29685, 29686,
 29687, 29688, 29689, 29690, 29691,
 29692, 29694, 29695, 29696, 29697,
 29698, 29699, 29700, 29701, 29702,
 29703, 29704, 29734, 29750, 29753,
 29754, 29755, 29760, 29762, 29763,
 29766, 29769, 29770, 29772, 29781,
 29785, 29787, 29797, 29801, 29804,
 29807, 29809, 29810, 29812, 29813,
 29814, 29816, 29818, 29819, 29821,
 29823, 29828, 29829, 29832, 29834,
 29837, 29838, 29840, 29850, 29852,
 29855, 29861, 29864, 29865, 29874,
 29887, 29889, 29905, 29908, 29918,
 29920, 29922, 29924, 29926, 29930,
 29932, 29934, 29936, 29941, 29952,
 29954, 29968, 29970, 29975, 29979,
 29980, 29987, 29996, 30000, 30003,
 30008, 30010, 30011, 30012, 30015,
 30016, 30017, 30018, 30033, 30035,
 30064, 30069, 30071, 30087, 30089,
 30118, 30130, 30139, 30142, 30149,
 30151, 30161, 30163, 30187, 30192,
 30194, 30195, 30204, 30207, 30211,
 30219, 30222, 30226, 30229, 30236,
 30252, 30254, 30255, 30257, 30258,
 30259, 30262, 30263, 30283, 30288,
 30292, 30309, 30318, 30325, 30333,
 30347, 30349, 30353, 30360, 30363,
 30364, 30365, 30367, 30370, 30375,
 30378, 30379, 30381, 30392, 30403,
 30405, 30406, 30407, 30411, 30413,
 30415, 30416, 30418, 30428, 30440,
 30454, 30455, 30458, 30463, 30487,
 30490, 30496, 30499, 30502, 30505,
 30508, 30510, 30512, 30516, 30517,
 30524, 30525, 30526, 30529, 30556,
 30606, 30616, 30629, 30634, 30647,
 30659, 30662, 30665, 30675, 30679,
 30695, 30699, 30725, 30728, 30729,
 30750, 30769, 30773, 30775, 30823,
 30833, 30844, 31021, 31056, 31068,
 31105, 31107, 31184, 31188, 31193,
 31196, 31200, 31202, 31216, 31246,
 31254, 31258, 31267, 31268, 31269,
 31276, 31280, 31282, 31295, 31296
 __fp_sep:B 28423
 __fp_sep:TF 28432
 __fp_set_function:Nnnn
 1287, 31025, 31027, 31030
 __fp_set_sign_o:w 25786,
 27021, 27803, 27804, 27804, 27826
 __fp_set_variable:nn
 30899, 30902, 30904
 __fp_show:NN
 26490, 26490, 26492, 26494
 __fp_show_validate:n
 26497, 26500, 26500
 __fp_show_validate:nn
 26500, 26502, 26511, 26513
 __fp_show_validate:w
 26500, 26521, 26538
 __fp_show_validate_aux:n 26500,

- 26509, 26546, 26554, 26564, 26565
 __fp_sign_aux_o:w
 27792, 27796, 27797, 27802
 __fp_sign_o:w .. 27025, 27792, 27792
 __fp_sin_o:w
 1080, 1123, 1254, 29108, 29108
 __fp_sin_series_aux_o:NNnwww ...
 29595, 29599, 29610
 __fp_sin_series_o:NNnwww
 1232, 1247, 29114,
 29129, 29144, 29159, 29595, 29595
 __fp_small_int:wTF
 ... 1228, 24554, 24554, 25004, 29029
 __fp_small_int_normal:NnwTF ...
 24554, 24558, 24570
 __fp_small_int_test:NnnwNwTF . 24554
 __fp_small_int_test:NnnwNw
 24573, 24576
 __fp_small_int_true:wTF
 .. 24554, 24557, 24562, 24569, 24579
 __fp_sqrt_auxi_o:NNNNwnnN
 27623, 27631, 27631
 __fp_sqrt_auxii_o:NnnnnnnnN ...
 1182, 1184,
 27633, 27637, 27637, 27717, 27729
 __fp_sqrt_auxiii_o:wnnnnnnnn ...
 27634, 27672, 27672, 27718
 __fp_sqrt_auxiv_o:NNNNNw
 27672, 27676, 27693
 __fp_sqrt_auxix_o:wnwnw
 27706, 27708, 27715
 __fp_sqrt_auxv_o:NNNNNw
 27672, 27680, 27695
 __fp_sqrt_auxvi_o:NNNNNw
 27672, 27684, 27697
 __fp_sqrt_auxvii_o:NNNNNw
 27672, 27687, 27699
 __fp_sqrt_auxviii_o:nnnnnnn ...
 27694,
 27696, 27698, 27704, 27706, 27706
 __fp_sqrt_auxx_o:Nnnnnnnn
 27702, 27720, 27720
 __fp_sqrt_auxxi_o:wnnnN
 27720, 27722, 27727
 __fp_sqrt_auxxii_o:nnnnnnnnw ...
 27730, 27734, 27734
 __fp_sqrt_auxxiii_o:w
 27734, 27741, 27754
 __fp_sqrt_auxxiv_o:wnnnnnnnN ...
 .. 27746, 27749, 27757, 27759, 27759
 __fp_sqrt_Newton_o:wnn ... 1182,
 27608, 27619, 27620, 27620, 27627
 __fp_sqrt_npos_auxi_o:wnnnN ...
 27599, 27605, 27610
 __fp_sqrt_npos_auxii_o:wnnnnnnnN
 27599, 27614, 27618
 __fp_sqrt_npos_o:w
 27596, 27599, 27599
 __fp_sqrt_o:w .. 27027, 27589, 27589
 __fp_step:NNnnnn
 26856, 26859, 26866, 26875
 __fp_step:NnnnnN ... 1152, 26790,
 26818, 26819, 26839, 26850, 26855
 __fp_step:wwwN . 26790, 26792, 26798
 __fp_step_fp:wwwN 26790, 26804, 26812
 __fp_str_if_eq:nn . 24591, 24591,
 25707, 25719, 26009, 26051, 28821
 __fp_sub_back_far_o:NnnwnnnnN ..
 1165, 27201, 27248, 27248
 __fp_sub_back_near_after:wnnnNw
 27207, 27209, 27216, 27286
 __fp_sub_back_near_o:nnnnnnnnN .
 1164, 27197, 27207, 27207
 __fp_sub_back_near_pack:NNNNNw
 27207, 27211, 27214, 27288
 __fp_sub_back_not_far_o:wwwnnN .
 27263, 27283, 27283
 __fp_sub_back_quite_far_ii:NN ..
 27267, 27269, 27273
 __fp_sub_back_quite_far_o:wwnnN .
 27261, 27267, 27267
 __fp_sub_back_shift:wnnnn
 1165, 27219, 27223, 27223
 __fp_sub_back_shift_ii:ww
 27223, 27225, 27228
 __fp_sub_back_shift_iii:NNNNNNNw
 27223, 27233, 27236, 27245
 __fp_sub_back_shift_iv:nnnnw ...
 27223, 27240, 27246
 __fp_sub_back_very_far_ii-
 o:nnNwNN 27295, 27298, 27302
 __fp_sub_back_very_far_o:wwwnnN
 27262, 27295, 27295
 __fp_sub_eq_o:Nnwnw
 27169, 27174, 27182
 __fp_sub_npos_i_o:Nnwnw
 ... 1163, 27176, 27186, 27190, 27190
 __fp_sub_npos_ii_o:Nnwnw
 27169, 27178, 27184
 __fp_sub_npos_o:NnwNnw
 1163, 27089, 27169, 27169
 __fp_symbolic_&o:ww 30681
 __fp_symbolic_&symbolic_o:ww 30681
 __fp_symbolic*_o:ww 30681
 __fp_symbolic_*symbolic_o:ww 30681
 __fp_symbolic+_o:ww 30681
 __fp_symbolic_+symbolic_o:ww 30681
 __fp_symbolic-_o:ww 30681

- __fp_symbolic_-_symbolic_o:ww [30681](#)
- __fp_symbolic/_o:ww [30681](#)
- __fp_symbolic/_symbolic_o:ww [30681](#)
- __fp_symbolic^_o:ww [30681](#)
- __fp_symbolic^symbolic_o:ww [30681](#)
- __fp_symbolic_acos_o:w [30701](#)
- __fp_symbolic_acsc_o:w [30701](#)
- __fp_symbolic_asec_o:w [30701](#)
- __fp_symbolic_asin_o:w [30701](#)
- __fp_symbolic_binary_o:Nww
- [30675](#), [30675](#), [30685](#)
- __fp_symbolic_binary_to_tl:Nww .
- [30749](#), [30755](#), [30769](#)
- __fp_symbolic_chk:w
 - . [1280](#), [1281](#), [26506](#), [26532](#), [26559](#),
 - [26563](#), [30629](#), [30629](#), [30634](#), [30647](#),
 - [30665](#), [30678](#), [30698](#), [30750](#), [30822](#),
 - [30827](#), [30843](#), [31012](#), [31053](#), [31062](#)
- __fp_symbolic_convert:wnnN
- [30713](#), [30717](#), [30725](#)
- __fp_symbolic_cos_o:w [30701](#)
- __fp_symbolic_cot_o:w [30701](#)
- __fp_symbolic_cs_arg_to_fn:NN ..
- [30731](#), [30731](#), [30765](#)
- __fp_symbolic_csc_o:w [30701](#)
- __fp_symbolic_exp_o:w [30701](#)
- __fp_symbolic_fact_o:w [30701](#)
- \l__fp_symbolic_flag [30899](#)
- \l__fp_symbolic_fp .. [1288](#), [30627](#),
- [30914](#), [30918](#), [30924](#), [31070](#), [31074](#)
- __fp_symbolic_function_to_tl:Nw
- [30749](#), [30756](#), [30778](#)
- __fp_symbolic_ln_o:w [30701](#)
- __fp_symbolic_not_o:w [30701](#)
- __fp_symbolic_op_arg_to_fn:nN ..
- [30731](#), [30733](#), [30736](#)
- __fp_symbolic_sec_o:w [30701](#)
- __fp_symbolic_set_sign_o:w .. [30701](#)
- __fp_symbolic_show_validate:w ..
- [26500](#), [26531](#), [26558](#)
- __fp_symbolic_sign_o:w [30701](#)
- __fp_symbolic_sin_o:w [30701](#)
- __fp_symbolic_tan_o:w [30701](#)
- __fp_symbolic_to_decimal:w .. [30713](#)
- __fp_symbolic_to_int:w [30713](#)
- __fp_symbolic_to_scientific:w [30713](#)
- __fp_symbolic_to_tl:w . [30749](#), [30749](#)
- __fp_symbolic_unary_o:NNw
- [30695](#), [30695](#), [30709](#)
- __fp_symbolic_unary_to_tl:NNw ..
- [30749](#), [30754](#), [30761](#)
- __fp_symbolic_|_o:ww [30681](#)
- __fp_symbolic_|_symbolic_o:ww [30681](#)
- __fp_tan_o:w [29168](#), [29168](#)
- __fp_tan_series_aux_o:Nnww ..
- [29666](#), [29670](#), [29679](#)
- __fp_tan_series_o:NNwww
 - ... [1234](#), [29175](#), [29190](#), [29666](#), [29666](#)
- __fp_ternary:NwwN
 - [1146](#), [26218](#), [26972](#), [26972](#)
- __fp_ternary_auxi:NwwN
 - [1146](#), [1156](#), [26972](#), [26981](#), [27001](#)
- __fp_ternary_auxii:NwwN
 - [1146](#), [1156](#), [26220](#), [26972](#), [26979](#), [27009](#)
- __fp_tmp:w
 - . [1076](#), [1133](#), [24495](#), [24505](#), [24506](#),
 - [24507](#), [24508](#), [24509](#), [24510](#), [24511](#),
 - [24512](#), [24513](#), [24514](#), [24515](#), [24516](#),
 - [24517](#), [24518](#), [24519](#), [24520](#), [24597](#),
 - [24599](#), [25112](#), [25124](#), [25125](#), [25126](#),
 - [25127](#), [25128](#), [25129](#), [25130](#), [25188](#),
 - [25210](#), [25769](#), [25786](#), [25787](#), [25843](#),
 - [25848](#), [25849](#), [25850](#), [25851](#), [25852](#),
 - [25853](#), [25857](#), [25865](#), [25866](#), [25867](#),
 - [25868](#), [25869](#), [25870](#), [25871](#), [25872](#),
 - [25873](#), [25874](#), [25875](#), [26079](#), [26095](#),
 - [26096](#), [26119](#), [26131](#), [26149](#), [26151](#),
 - [26153](#), [26155](#), [26157](#), [26159](#), [26161](#),
 - [26163](#), [26167](#), [26181](#), [26182](#), [26197](#),
 - [26198](#), [26199](#), [26217](#), [26219](#), [27842](#),
 - [27856](#), [27857](#), [30681](#), [30694](#), [30713](#),
 - [30722](#), [30723](#), [30724](#), [30840](#), [30851](#),
 - [30857](#), [30861](#), [30986](#), [30992](#), [30998](#),
 - [31002](#), [31073](#), [31078](#), [31082](#), [31086](#)
- __fp_to_decimal:w [30084](#),
- [30094](#), [30094](#), [30211](#), [30228](#), [31247](#)
- __fp_to_decimal_dispatch:w
 - . [1260](#), [1263](#), [1264](#), [26849](#), [30074](#),
 - [30078](#), [30081](#), [30081](#), [30093](#), [30722](#)
- __fp_to_decimal_huge:wnnnn
 - [30094](#), [30129](#), [30151](#)
- __fp_to_decimal_large:Nnnw
 - [30094](#), [30125](#), [30142](#)
- __fp_to_decimal_normal:wnnnnn ..
 - [30094](#), [30099](#), [30117](#), [30182](#)
- __fp_to_decimal_recover:w
 - [30081](#), [30084](#), [30087](#)
- __fp_to_dim:w .. [30196](#), [30206](#), [30211](#)
- __fp_to_dim_dispatch:w
 - ... [1264](#), [30196](#), [30197](#), [30201](#), [30204](#)
- __fp_to_dim_recover:w
 - [30196](#), [30206](#), [30209](#)
- __fp_to_int:w ... [1264](#), [30221](#), [30226](#)
- __fp_to_int_dispatch:w
 - .. [30212](#), [30212](#), [30216](#), [30219](#), [30723](#)
- __fp_to_int_recover:w
 - [30212](#), [30221](#), [30224](#)


```

\__fp_to_scientific:w .....
    ..... 1261, 30030, 30040, 30040
\__fp_to_scientific_dispatch:w ..
    ..... 1259, 1263, 30020,
    30024, 30027, 30027, 30039, 30724
\__fp_to_scientific_normal:wnnnnn
    ..... 30040, 30045, 30063
\__fp_to_scientific_normal:wNw ..
    ..... 30040, 30066, 30071
\__fp_to_scientific_recover:w ...
    ..... 30027, 30030, 30033
\__fp_to_tl:w .....
    ..... 30160, 30168, 30168, 31255
\__fp_to_tl_dispatch:w 1258, 1262,
    30152, 30156, 30159, 30159, 30167,
    30292, 30633, 30766, 30773, 30775
\__fp_to_tl_normal:nnnnn .....
    ..... 30168, 30173, 30178
\__fp_to_tl_recover:w .....
    ..... 30159, 30160, 30161
\__fp_to_tl_scientific:wnnnnn ...
    ..... 30168, 30183, 30186
\__fp_to_tl_scientific:wNw .....
    ..... 30168, 30189, 30194
\c__fp_trailing_shift_int .....
    ..... 24439, 27874,
    27897, 27971, 28901, 29541, 29578
\__fp_trap_division_by_zero_-
    set:N .....
    .. 24671, 24672, 24674, 24676, 24677
\__fp_trap_division_by_zero_set_-
    error: ..... 24671, 24671
\__fp_trap_division_by_zero_set_-
    flag: ..... 24671, 24673
\__fp_trap_division_by_zero_set_-
    none: ..... 24671, 24675
\__fp_trap_invalid_operation_-
    set:N .....
    .. 24635, 24636, 24638, 24640, 24641
\__fp_trap_invalid_operation_-
    set_error: ..... 24635, 24635
\__fp_trap_invalid_operation_-
    set_flag: ..... 24635, 24637
\__fp_trap_invalid_operation_-
    set_none: ..... 24635, 24639
\__fp_trap_overflow_set:N .....
    .. 24701, 24702, 24704, 24706, 24707
\__fp_trap_overflow_set:NnNn ...
    ..... 24701, 24708, 24716, 24717
\__fp_trap_overflow_set_error: ..
    ..... 24701, 24701
\__fp_trap_overflow_set_flag: ...
    ..... 24701, 24703

\__fp_trap_overflow_set_none: ...
    ..... 24701, 24705
\__fp_trap_underflow_set:N .....
    .. 24701, 24710, 24712, 24714, 24715
\__fp_trap_underflow_set_error: .
    ..... 24701, 24709
\__fp_trap_underflow_set_flag: ..
    ..... 24701, 24711
\__fp_trap_underflow_set_none: ..
    ..... 24701, 24713
\__fp_trig:NNNNwn .....
    ..... 29114, 29129, 29144,
    29159, 29174, 29189, 29206, 29206
\c__fp_trig_intarray ..... 1242,
    29267, 29497, 29500, 29503, 29506,
    29509, 29512, 29515, 29518, 29521
\__fp_trig_large:ww .....
    ..... 29214, 29481, 29481
\__fp_trig_large_auxi:w .....
    ..... 29481, 29483, 29488
\__fp_trig_large_auxii:w .....
    ..... 1242, 29481, 29491, 29525
\__fp_trig_large_auxiii:w . 1242,
    29481, 29499, 29502, 29505, 29508,
    29511, 29514, 29517, 29520, 29533
\__fp_trig_large_auxix:Nw .....
    ..... 29554, 29564, 29567, 29571
\__fp_trig_large_auxv:www .....
    ..... 29531, 29534, 29534
\__fp_trig_large_auxvi:wnnnnnnnn
    ..... 29534, 29540, 29545
\__fp_trig_large_auxvii:w .....
    ..... 29537, 29554, 29554
\__fp_trig_large_auxviii:w ... 29554
\__fp_trig_large_auxviii:ww ...
    ..... 29556, 29560
\__fp_trig_large_auxx:wNNNNN ...
    ..... 29554, 29577, 29581
\__fp_trig_large_auxxi:w .....
    ..... 29554, 29574, 29588
\__fp_trig_large_pack:NNNNw ...
    ..... 29534, 29547, 29552, 29583
\__fp_trig_small:ww .. 1236, 1244,
    29216, 29220, 29220, 29226, 29593
\__fp_trig_large:ww .....
    ..... 29214, 29228, 29228
\__fp_trig_large_auxi:nnnnwNNNN
    ..... 29228, 29234, 29240
\__fp_trig_large_auxii:wNw ...
    ..... 29228, 29242, 29248
\__fp_trig_large_auxiii:www ...
    ..... 29228, 29251, 29255
\__fp_trig_small:ww .....
    ... 1236, 29216, 29222, 29222, 29265

```


__fp_trim_zeros:w
 .. [30008](#), [30008](#), [30135](#), [30144](#), [30195](#)
 __fp_trim_zeros_dot:w
 [30008](#), [30012](#), [30016](#)
 __fp_trim_zeros_end:w
 [30008](#), [30017](#), [30018](#)
 __fp_trim_zeros_loop:w
 [30008](#), [30010](#), [30011](#), [30015](#)
 __fp_tuple [26961](#), [26962](#), [26965](#), [26967](#)
 __fp_tuple_&o:ww [26944](#)
 __fp_tuple_&_tuple_o:ww [26944](#)
 __fp_tuple*_o:ww [27830](#)
 __fp_tuple+_tuple_o:ww [27842](#)
 __fp_tuple-_tuple_o:ww [27842](#)
 __fp_tuple/_o:ww [27830](#)
 __fp_tuple_chk:w
 [1069](#), [24342](#), [24343](#),
 [24344](#), [24346](#), [24348](#), [24349](#), [24426](#),
 [24429](#), [26128](#), [26340](#), [26355](#), [26380](#),
 [26383](#), [26399](#), [26400](#), [26403](#), [26505](#),
 [26527](#), [26550](#), [26554](#), [26688](#), [26689](#),
 [27845](#), [27846](#), [27852](#), [27853](#), [29987](#)
 __fp_tuple_compare_back:ww
 [26685](#), [26686](#)
 __fp_tuple_compare_back_loop:w .
 [26685](#), [26695](#), [26703](#), [26712](#)
 __fp_tuple_compare_back_-
 tuple:ww [26685](#), [26687](#)
 __fp_tuple_convert:Nw
 .. [29987](#), [29987](#), [30039](#), [30093](#), [30167](#)
 __fp_tuple_convert_end:w
 [29987](#), [29992](#), [29996](#), [30006](#)
 __fp_tuple_convert_loop:nNw ...
 [29987](#), [29995](#), [30000](#), [30003](#)
 __fp_tuple_count:w
 [24347](#), [24348](#), [24349](#)
 __fp_tuple_count_loop:Nw
 [24347](#), [24352](#), [24356](#), [24357](#)
 __fp_tuple_map_loop_o:nw
 [26380](#), [26386](#), [26391](#), [26396](#)
 __fp_tuple_map_o:nw [26380](#),
 [26380](#), [27823](#), [27831](#), [27834](#), [27839](#)
 __fp_tuple_mapthread_loop_o:nw .
 [26398](#), [26406](#), [26412](#), [26418](#)
 __fp_tuple_mapthread_o:nww
 [26398](#), [26398](#), [27850](#)
 __fp_tuple_not_o:w ... [26935](#), [26943](#)
 __fp_tuple_set_sign_aux_o:Nnw ..
 [27814](#), [27817](#), [27822](#)
 __fp_tuple_set_sign_aux_o:w ...
 [27814](#), [27823](#), [27824](#)
 __fp_tuple_set_sign_o:w [27814](#), [27814](#)
 __fp_tuple_show_validate:w
 [26500](#), [26526](#), [26549](#)
 __fp_tuple_to_decimal:w [30081](#), [30092](#)
 __fp_tuple_to_scientific:w
 [30027](#), [30038](#)
 __fp_tuple_to_tl:w ... [30159](#), [30166](#)
 __fp_tuple_l_o:ww [26944](#)
 __fp_tuple_l_tuple_o:ww [26944](#)
 __fp_type_from_scan:N [1070](#),
 [24371](#), [24371](#), [25954](#), [25956](#), [25980](#),
 [25982](#), [25993](#), [25995](#), [26633](#), [26635](#),
 [26649](#), [26651](#), [30591](#), [30611](#), [30613](#)
 __fp_type_from_scan:w
 [24371](#), [24380](#), [24385](#)
 __fp_type_from_scan_other:N ...
 .. [24371](#), [24375](#), [24378](#), [24395](#), [24413](#)
 __fp_types_binary:Nww
 [1278](#), [1280](#), [30601](#), [30601](#), [30679](#), [30755](#)
 __fp_types_binary_auxi:Nww
 [30601](#), [30603](#), [30606](#)
 __fp_types_binary_auxii:NNww ...
 [30601](#), [30608](#), [30618](#)
 __fp_types_cs_to_op:N
 [30568](#), [30568](#),
 [30586](#), [30604](#), [30734](#), [30774](#), [30782](#)
 __fp_types_cs_to_op_auxi:wwwn ..
 [30568](#), [30570](#), [30579](#)
 __fp_types_unary:NNw
 [1278](#), [1280](#), [30583](#), [30583](#), [30699](#), [30754](#)
 __fp_types_unary_auxi:nNw
 [30583](#), [30585](#), [30588](#)
 __fp_types_unary_auxii:NnNw ...
 [30583](#), [30590](#), [30595](#)
 __fp_underflow:w [1068](#),
 [1082](#), [1084](#), [24291](#), [24732](#), [24738](#), [28602](#)
 __fp_use_i:ww
 [1199](#), [1255](#), [24227](#), [24227](#), [28090](#), [29916](#)
 __fp_use_i:www [24227](#), [24228](#)
 __fp_use_i_delimit_by_s_stop:nw
 [24238](#),
 [24238](#), [26606](#), [26977](#), [30572](#), [30574](#)
 __fp_use_i_until_s:nw
 [1244](#), [24220](#), [24221](#), [24271](#),
 [24281](#), [24546](#), [29258](#), [29536](#), [29542](#),
 [29573](#), [30353](#), [30424](#), [31097](#), [31193](#)
 __fp_use_ii_until_s:nnw
 [24220](#), [24222](#), [24269](#), [24280](#)
 __fp_use_none_stop_f:n
 .. [24217](#), [24217](#), [28261](#), [28262](#), [28263](#)
 __fp_use_none_until_s:w . [24220](#),
 [24220](#), [27625](#), [28962](#), [29911](#), [29914](#)
 __fp_use_s:n [24218](#), [24218](#)
 __fp_use_s:nn [24218](#), [24219](#)
 __fp_variable_o:w [1281](#),
 [30811](#), [30811](#), [30823](#), [30828](#), [30844](#)

- _fp_variable_set_parsing:Nn . . .
 . . . [30838](#), [30838](#), [30878](#), [30896](#), [30912](#)
- _fp_variable_set_parsing-
 aux:NNn [30838](#), [30846](#), [30849](#)
- _fp_zero_fp:N
 [24262](#), [24262](#), [24716](#), [25077](#)
- _fp_l_o:ww [1146](#), [26944](#)
- _fp_l_symbolic_o:ww [30681](#)
- _fp_l_tuple_o:ww [26944](#)
- farray commands:
- \fparray_count:N . . . [290](#), [291](#), [31153](#),
 [31153](#), [31158](#), [31165](#), [31176](#), [31232](#)
- \fparray_gset:Nnn
 [290](#), [1296](#), [31178](#), [31178](#), [31187](#)
- \fparray_gzero:N
 [290](#), [31229](#), [31229](#), [31241](#)
- \fparray_if_exist:N . . . [31297](#), [31299](#)
- \fparray_if_exist:NTF . . . [291](#), [31297](#)
- \fparray_if_exist_p:N . . . [291](#), [31297](#)
- \fparray_item:Nn
 [291](#), [1296](#), [31242](#), [31242](#), [31249](#)
- \fparray_item_to_tl:Nn
 [291](#), [31242](#), [31250](#), [31257](#)
- \fparray_new:Nn
 [290](#), [31126](#), [31126](#), [31138](#)
- \futurelet [237](#)
- G**
- \gdef [238](#)
- get commands:
- get_luaadata [12163](#)
- \GetIdInfo [11](#), [11660](#)
- \gleaders [832](#)
- \glet [833](#)
- \global [105](#), [140](#), [239](#)
- \globaldefs [240](#)
- \glueexpr [494](#)
- \glueshrink [495](#)
- \glueshrinkorder [496](#)
- \gluestretch [497](#)
- \gluestretchorder [498](#)
- \gluetomu [499](#)
- \glyphdimensionsmode [834](#)
- graphics commands:
- \l_graphics_ext_type_prop
 [340](#), [40593](#), [40627](#)
- \graphics_get_full_name:nN
 [340](#), [40701](#), [40701](#), [40706](#)
- \graphics_get_full_name:nTF
 [340](#), [40701](#), [40703](#)
- \graphics_get_pagecount:nN
 [340](#), [40744](#), [40744](#)
- \graphics_get_pagecount:nn [340](#)
- \graphics_include:nn
 [340](#), [40596](#), [40596](#), [40610](#)
- \graphics_log_list: [341](#), [40691](#), [40692](#)
- \l_graphics_search_ext_seq
 [340](#), [40592](#), [40718](#), [40736](#)
- \l_graphics_search_path_seq
 [340](#), [40591](#), [40600](#), [40710](#), [40747](#)
- \graphics_show_list: [341](#), [40691](#), [40691](#)
- graphics internal commands:
- _graphics_backend_get_pagecount:n
 [40752](#)
- _graphics_bb_restore:nTF
 [40461](#), [40475](#), [40490](#), [40535](#)
- _graphics_bb_save:n
 [40461](#), [40461](#), [40474](#), [40549](#)
- \l_graphics_decodearray_str [40433](#)
- \l_graphics_dir_str
 [40588](#), [40616](#), [40717](#)
- \l_graphics_dir_str\l_-
 graphics_name_str\l_-
 graphics_ext_str [40588](#)
- \l_graphics_draft_bool [40433](#), [40645](#)
- \l_graphics_ext_str
 [40590](#), [40616](#), [40620](#), [40717](#), [40720](#)
- _graphics_extract_bb:n [40491](#), [40491](#)
- _graphics_extract_bb_aux:nn [40491](#)
- _graphics_extract_bb_auxi:nn
 [40494](#), [40497](#), [40499](#)
- _graphics_extract_bb_auxii:nnn
 [40491](#), [40495](#), [40498](#), [40500](#)
- _graphics_extract_bb_auxiii:nnnn
 [40491](#), [40504](#), [40507](#), [40509](#)
- _graphics_extract_bb_auxiv:nnn
 [40491](#), [40503](#), [40508](#), [40510](#)
- \l_graphics_final_name_str
 [40585](#), [40637](#), [40642](#), [40644](#)
- \l_graphics_full_name_str [40585](#),
 [40601](#), [40603](#), [40615](#), [40637](#), [40638](#),
 [40639](#), [40641](#), [40668](#), [40682](#), [40711](#),
 [40713](#), [40716](#), [40728](#), [40735](#), [40738](#),
 [40742](#), [40748](#), [40750](#), [40753](#), [40755](#)
- _graphics_get_full_name:n
 [40701](#), [40714](#), [40723](#), [40726](#), [40733](#)
- _graphics_get_pagecount:n
 [40744](#), [40764](#)
- _graphics_get_pagecount:nw
 [40744](#), [40773](#), [40783](#)
- _graphics_include:
 [40596](#), [40605](#), [40611](#)
- _graphics_include_auxi:n
 [40596](#), [40617](#), [40623](#), [40625](#), [40631](#)
- _graphics_include_auxii:n
 [40596](#), [40629](#), [40632](#)

- __graphics_include_auxiii:n ...
..... 40596, 40646, 40652
- __graphics_include_auxiv:n
..... 40596, 40647, 40677
- __graphics_include_search:n . 40596
- \l__graphics_interpolate_bool . 40433
- __graphics_list:N
..... 40691, 40691, 40692, 40693
- __graphics_list_aux:n
..... 40691, 40697, 40700
- \l__graphics_llx_dim
40457, 40466, 40467, 40482, 40483,
40579, 40654, 40662, 40665, 40688
- \l__graphics_lly_dim
..... 40457, 40468, 40469,
40485, 40486, 40580, 40659, 40686
- \l__graphics_name_str
..... 40589, 40616, 40717
- \l__graphics_page_int
..... 40433, 40493, 40494
- \l__graphics_pagebox_str 1524
- \l__graphics_pagebox_tl
..... 40433, 40502, 40504
- \l__graphics_pdf_str 40433
- __graphics_read_bb:n . 40491, 40516
- __graphics_read_bb_auxi:nnnn ...
..... 40491, 40512, 40518, 40533
- __graphics_read_bb_auxii:nnnn ..
..... 40491, 40536, 40538
- __graphics_read_bb_auxii:w . 40491
- __graphics_read_bb_auxiii:w ...
..... 40546, 40555
- __graphics_read_bb_auxiv:w
..... 40491, 40561, 40564
- __graphics_read_bb_auxv:w
..... 40491, 40574, 40577
- \g__graphics_record_seq
..... 40595, 40642, 40695, 40697
- \l__graphics_tmp_box 40587,
40679, 40684, 40685, 40687, 40689
- \l__graphics_tmp_dim 40429
- \l__graphics_tmp_ior
40429, 40513, 40519, 40541, 40544,
40551, 40766, 40768, 40771, 40779
- \l__graphics_tmp_tl 40429,
40568, 40574, 40627, 40628, 40629
- \l__graphics_type_str
..... 40433, 40613, 40623
- \l__graphics_urx_dim
..... 40457, 40470, 40479,
40581, 40654, 40662, 40665, 40688
- \l__graphics_ury_dim 40457,
40471, 40480, 40582, 40659, 40686
- group commands:
- \group_align_safe_begin/end: .. 615
- \group_align_safe_begin:
... 75, 608, 727, 732, 3662, 4105,
8478, 8705, 8707, 12705, 13348,
20094, 20559, 20580, 20612, 25907,
25922, 32997, 33421, 35518, 39276
- \group_align_safe_end: 75,
727, 732, 3665, 4117, 8480, 8705,
8710, 12726, 13331, 20138, 20568,
20577, 20617, 20623, 25908, 25923,
33009, 33428, 35529, 39279, 39283
- \group_begin: 14, 721, 1480,
1499, 1415, 1416, 2297, 2300, 2303,
2742, 2937, 3124, 3292, 3329, 3608,
3661, 3668, 3688, 3765, 3931, 4124,
4189, 4559, 4651, 4978, 5487, 5821,
6062, 6163, 6590, 6965, 7245, 7503,
7529, 7541, 7551, 7560, 7746, 7775,
7897, 8705, 8754, 8977, 9073, 9209,
9221, 9419, 9443, 9459, 9524, 10450,
10699, 10745, 11008, 11151, 11667,
12298, 12484, 12638, 12642, 12743,
12748, 12827, 12840, 13691, 14012,
14035, 14540, 14650, 14705, 15000,
15048, 15094, 15101, 15431, 15613,
17514, 17519, 17725, 17756, 18395,
18407, 19968, 19974, 20022, 20084,
20141, 20159, 20183, 20268, 20287,
20695, 21928, 22225, 22354, 22396,
22439, 23587, 26944, 31040, 31590,
31799, 31824, 32131, 32490, 32685,
32955, 34867, 34900, 35739, 35801,
36522, 38627, 38803, 39407, 39498,
39539, 40598, 40709, 40746, 42001,
42004, 42066, 42416, 42489, 42492
- \c_group_begin_token
..... 117, 206, 214, 742, 938,
1429, 3730, 4281, 13192, 13232,
20118, 20141, 20165, 32747, 36565,
36571, 36585, 36591, 36621, 36623,
36642, 36690, 36703, 36765, 36771,
36781, 36787, 39335, 39336, 39343
- \group_end: 14, 15, 481, 482,
590, 867, 1309, 1313, 1480, 1415,
1417, 2297, 2300, 2306, 2751, 2940,
3127, 3296, 3338, 3547, 3621, 3666,
3687, 3711, 3772, 3955, 4111, 4211,
4571, 4665, 5011, 5019, 5500, 5825,
6122, 6170, 6177, 6185, 6594, 6595,
7002, 7309, 7508, 7536, 7624, 7769,
7816, 7898, 7899, 8712, 8773, 8994,
9101, 9213, 9225, 9438, 9451, 9470,
9670, 10456, 10703, 10774, 11031,

- 11169, 11670, 12301, 12506, 12556,
 12604, 12626, 12645, 12649, 12753,
 12761, 12830, 12844, 13709, 14017,
 14040, 14550, 14665, 14708, 15013,
 15060, 15119, 15181, 15612, 15744,
 17524, 17534, 17737, 17766, 17771,
 18399, 18411, 19976, 19983, 20083,
 20088, 20158, 20162, 20190, 20286,
 20335, 20719, 21944, 22351, 22377,
 22436, 22443, 23601, 26969, 31085,
 31594, 31803, 31855, 32463, 32571,
 32701, 32975, 34893, 34926, 35743,
 36046, 36528, 38628, 38808, 39411,
 39503, 39553, 40608, 40727, 40761,
 42020, 42077, 42487, 42506, 42726
 \c_group_end_token 206,
 938, 3733, 20121, 20141, 20170,
 32748, 36579, 36628, 39339, 39347
 \group_insert_after:N
 15, 1421, 1421, 4567,
 36643, 36704, 38635, 39339, 39340,
 39371, 39655, 40825, 40830, 40835
 \group_log_list: 15, 2309, 2311
 \group_show_list: 15, 2309, 2309
 groups commands:
 .groups:n 249, 23022
 \gtoksapp 835
 \gtokspre 836
- ## H
- \H 65, 33333, 35799,
 35819, 35966, 35967, 35994, 35995
 \halign 241
 \hangingafter 242
 \hangindent 243
 \hbadness 244
 \hbox 245
 hbox commands:
 \hbox:n ... 311, 315, 36536, 36536,
 36854, 37151, 37192, 37233, 38414
 \hbox_gset:Nn
 315, 36538, 36543, 36549,
 36821, 36944, 36988, 37008, 37028,
 37045, 37066, 37095, 37106, 37177,
 37220, 37244, 37453, 37897, 42244
 \hbox_gset:Nw
 315, 36562, 36568, 36575, 37526, 42246
 \hbox_gset_end:
 315, 36562, 36581, 37529
 \hbox_gset_to_wd:Nnn
 315, 36550, 36555, 36561, 42245
 \hbox_gset_to_wd:Nnw
 316, 36582, 36588, 36595, 42247
 \hbox_overlap_center:n
 315, 36606, 36606
 \hbox_overlap_left:n 315, 36606, 36608
 \hbox_overlap_right:n
 315, 36606, 36610
 \hbox_set:Nn . 311, 315, 331, 36538,
 36538, 36548, 36818, 36850, 36851,
 36938, 36985, 37005, 37025, 37042,
 37063, 37092, 37100, 37124, 37171,
 37213, 37241, 37254, 37262, 37270,
 37279, 37288, 37305, 37313, 37321,
 37327, 37340, 37440, 37894, 37917,
 38174, 38261, 38547, 40679, 42162
 \hbox_set:Nw
 315, 36562, 36562, 36574, 37513, 42164
 \hbox_set_end:
 315, 316, 36562, 36576, 36581, 37516
 \hbox_set_to_wd:Nnn
 315, 316, 36550, 36550, 36560, 42163
 \hbox_set_to_wd:Nnw
 316, 36582, 36582, 36594, 42165
 \hbox_to_wd:nn
 315, 36596, 36596, 37142, 40654, 40664
 \hbox_to_zero:n 315, 36596,
 36601, 36607, 36609, 36611, 41130
 \hbox_unpack:N
 316, 36612, 36612, 36614, 38178
 \hbox_unpack_drop:N
 319, 36612, 36613, 36615
 hcoffin commands:
 \hcoffin_gset:Nn
 326, 37436, 37449, 37461
 \hcoffin_gset:Nw
 326, 37509, 37522, 37534
 \hcoffin_gset_end:
 326, 37509, 37527, 37536
 \hcoffin_set:Nn
 326, 327, 37436, 37436,
 37448, 38418, 38425, 38463, 38499
 \hcoffin_set:Nw
 326, 37509, 37509, 37521
 \hcoffin_set_end:
 326, 37509, 37514, 37535
 \hfi 1144
 \hfil 246
 \hfill 247
 \hfilneg 248
 \hfuzz 249
 \hjcode 827
 \hoffset 250
 \holdinginserts 251
 hook commands:
 \hook_gput_code:nnn ... 31726, 31728
 \hpack 828

- \hrule 252
 - \hsize 253
 - \hskip 254
 - \hss 255
 - \ht 256
 - \hyphenation 257
 - \hyphenationbounds 829
 - \hyphenationmin 830
 - \hyphenchar 258
 - \hyphenpenalty 259
 - \hyphenpenaltymode 831
- I**
- \i 34891,
35774, 35875, 35877, 35879, 35881,
35932, 35935, 35938, 35941, 36012
 - \if 260
 - if commands:
 - \if:w 29, 30, 201,
404, 405, 438, 507, 712, 730, 731,
733, 744, 745, 762, 944, 1386, 1392,
1802, 2167, 2168, 2829, 2832, 2833,
2834, 2835, 2850, 2851, 2852, 2853,
2854, 2855, 2856, 2857, 2858, 2922,
2923, 2925, 4806, 4835, 4889, 11194,
12788, 12798, 12891, 13246, 13266,
13281, 13831, 13838, 13843, 18957,
20338, 20341, 20357, 20363, 20480,
21950, 21965, 21966, 25006, 25379,
25383, 25405, 25499, 25531, 25550,
25616, 25630, 25647, 25668, 25707,
25719, 26009, 26051, 26171, 26186,
26591, 28821, 28851, 30365, 32494,
32503, 32532, 32731, 41855, 41867,
41869, 41950, 41951, 41952, 41953,
41967, 41968, 41978, 41979, 41980,
41981, 41982, 41983, 41984, 41985,
41986, 42755, 42757, 42761, 42763
 - \if_bool:N .. 74, 603, 1396, 8366, 8417
 - \if_box_empty:N
..... 323, 36474, 36476, 36486
 - \if_case:w 185, 769,
771, 811, 898, 1077, 1171, 1227,
1272, 2074, 3741, 3941, 4135, 4518,
4790, 5604, 5633, 5690, 6362, 6415,
7034, 7081, 7179, 7496, 8007, 8018,
10860, 14111, 14185, 14526, 15558,
18199, 18203, 18848, 18881, 20074,
24286, 24541, 24556, 24943, 24973,
26259, 26300, 27036, 27172, 27250,
27275, 27328, 27778, 27794, 27811,
28100, 28341, 28368, 28536, 28571,
28729, 28776, 28827, 28955, 28978,
29011, 29070, 29110, 29125, 29140,
29155, 29170, 29185, 29738, 29791,
29876, 29891, 29943, 29956, 30043,
30097, 30171, 30362, 31207, 31286
 - \if_catcode:w .. 30, 742, 937, 952,
1386, 1394, 2969, 3730, 3733, 3888,
3890, 3892, 3894, 3896, 3898, 3900,
4136, 4137, 4281, 13187, 13230,
20097, 20100, 20103, 20106, 20109,
20112, 20115, 20118, 20121, 20124,
20165, 20170, 20175, 20180, 20187,
20194, 20199, 20204, 20209, 20214,
20219, 20226, 20253, 20353, 20590,
20649, 20654, 20701, 20702, 24183,
25134, 25339, 25658, 25705, 26008,
26050, 32689, 32690, 32732, 32747,
32748, 32749, 32750, 32751, 32752,
32753, 32754, 32755, 32778, 32781,
32784, 32787, 32790, 32793, 32796
 - \if_charcode:w
..... 30, 201, 473, 742, 773, 952,
1386, 1393, 3800, 3824, 3873, 4177,
4214, 4216, 4727, 4737, 5248, 5803,
6971, 6974, 11456, 11465, 13173,
13223, 14271, 14506, 15170, 16646,
20231, 20651, 24544, 26604, 26975
 - \if_cs_exist:N 29, 30, 1401, 1401,
1830, 1868, 2745, 20025, 20261, 20489
 - \if_cs_exist:w ... 30, 1401, 1402,
1429, 1840, 1853, 1882, 1897, 1922,
1933, 2062, 4322, 11186, 19103,
19128, 19139, 21092, 21139, 22502
 - \if_dim:w 244, 21580,
21580, 21669, 21681, 21728, 21899
 - \if_eof:w 102,
663, 10393, 10393, 10398, 10483, 10501
 - \if_false: 29, 68, 214,
464, 486, 572, 584, 585, 588, 615,
684, 721, 728, 732, 741, 870, 887,
935, 977, 1386, 1387, 1857, 1863,
3650, 3719, 3768, 3771, 4285, 4286,
4293, 4294, 4987, 5006, 5007, 5016,
5081, 5124, 5138, 5142, 5354, 5387,
5399, 5403, 5437, 5442, 5450, 5485,
5492, 5497, 5545, 5782, 5801, 5812,
5835, 5847, 5848, 5851, 7261, 7278,
7615, 7654, 7662, 7669, 7699, 7824,
7826, 7827, 7833, 8708, 8711, 8978,
8986, 10839, 10879, 10883, 10890,
10898, 11150, 11163, 12226, 12230,
12485, 12492, 12721, 12722, 12862,
12866, 12906, 13151, 13156, 13247,
13260, 13278, 13282, 13292, 13613,
13625, 13669, 13679, 17602, 17605,
17844, 17849, 18438, 20047, 20053,

20071, 20350, 20364, 21393, 21394,
 21395, 21396, 21431, 21432, 21433,
 21434, 21715, 22572, 22584, 31367
 \if_hbox:N . . . 323, 36474, 36474, 36478
 \if_int_compare:w 29, 185,
 762, 887, 888, 1419, 1419, 3217,
 3274, 3303, 3345, 3356, 3359, 3377,
 3432, 3442, 3452, 3681, 3703, 3777,
 3810, 3818, 3841, 3865, 3922, 3937,
 4030, 4033, 4228, 4403, 4460, 4466,
 4467, 4474, 4478, 4484, 4485, 4490,
 4491, 4499, 4500, 4501, 4506, 4537,
 4538, 4787, 4808, 4809, 4810, 4813,
 4817, 4818, 4821, 4822, 4837, 4838,
 4841, 4845, 4846, 4849, 4910, 4928,
 4938, 4947, 4955, 4957, 4967, 4970,
 4998, 5085, 5197, 5261, 5266, 5294,
 5352, 5385, 5496, 5513, 5859, 5892,
 5923, 6309, 6380, 6406, 6467, 6480,
 6491, 6507, 6558, 6599, 6605, 6611,
 6775, 6776, 6803, 6830, 6929, 6986,
 7105, 7114, 7125, 7140, 7196, 7205,
 7257, 7274, 7297, 7563, 7592, 7650,
 7658, 7727, 10396, 10397, 10846,
 11472, 13439, 13448, 13497, 13498,
 13504, 13819, 13826, 14095, 14150,
 14151, 14157, 14169, 14185, 14321,
 14515, 14523, 14732, 14733, 14734,
 14739, 14740, 14764, 14816, 14916,
 15135, 15197, 15201, 15231, 15234,
 15250, 15254, 15276, 15356, 15358,
 15377, 15378, 15396, 15398, 15452,
 15455, 15456, 15574, 15575, 15716,
 15721, 16633, 18199, 18255, 18296,
 18297, 18418, 18471, 18473, 18475,
 18477, 18479, 18481, 18483, 18486,
 18518, 18654, 19999, 20000, 20001,
 20002, 20007, 20008, 20012, 20471,
 21744, 24039, 24042, 24086, 24150,
 24169, 24287, 24288, 24483, 24581,
 24823, 24833, 24841, 24854, 24867,
 24874, 24895, 24907, 24916, 24927,
 25039, 25044, 25116, 25146, 25299,
 25301, 25338, 25343, 25396, 25416,
 25443, 25457, 25493, 25520, 25548,
 25564, 25580, 25598, 25658, 25678,
 25694, 25778, 25801, 25830, 25832,
 26017, 26019, 26059, 26061, 26083,
 26100, 26105, 26135, 26203, 26248,
 26615, 26673, 26676, 26707, 26716,
 26719, 26724, 26725, 26728, 26731,
 26914, 27040, 27061, 27098, 27196,
 27251, 27252, 27255, 27258, 27329,
 27338, 27549, 27622, 27675, 27679,
 27683, 27701, 27736, 27737, 27738,
 27739, 27740, 27766, 28103, 28106,
 28202, 28297, 28354, 28370, 28506,
 28541, 28599, 28608, 28648, 28822,
 28833, 28851, 28875, 28907, 28910,
 28958, 28988, 29035, 29049, 29213,
 29257, 29742, 29780, 29789, 29825,
 29910, 29913, 30145, 30352, 30420,
 30421, 30422, 30432, 30460, 30465,
 30466, 30532, 30533, 30534, 30538,
 30553, 30558, 31094, 31161, 31165,
 31369, 32009, 32010, 32016, 32495,
 32720, 32764, 32881, 32888, 32905,
 32908, 34248, 34251, 34252, 34255,
 34256, 34277, 34280, 34283, 34286,
 34289, 34292, 34311, 34312, 34318,
 34321, 34324, 34327, 34330, 34333,
 34336, 34339, 34342, 34345, 34348,
 34351, 34354, 34357, 34360, 34389,
 34392, 34407, 34410, 34424, 34427,
 34430, 34433, 34449, 34452, 34455
 \if_int_odd:w 186,
 1247, 3947, 4527, 4918, 4926, 4936,
 5358, 5673, 18199, 18202, 18330,
 18556, 18564, 19072, 19998, 20006,
 20700, 24845, 24892, 24904, 26296,
 27312, 27604, 28999, 29563, 29602,
 29612, 29672, 29706, 29867, 30531
 \if_meaning:w 30, 482,
 742, 850, 864, 1155, 1339, 1386,
 1395, 1651, 1677, 1695, 1759, 1764,
 1773, 1826, 1848, 1864, 1872, 1892,
 1908, 1943, 2092, 2106, 2252, 2430,
 2486, 2487, 2777, 2800, 2809, 3060,
 3072, 3073, 3227, 3228, 3693, 3705,
 3727, 3757, 3785, 3886, 4098, 4138,
 4139, 4169, 4239, 4278, 4320, 4566,
 4722, 4747, 4759, 4888, 4909, 5245,
 5247, 5680, 6344, 6515, 6526, 6541,
 6702, 6745, 6880, 7152, 7474, 7591,
 7704, 8491, 8513, 10795, 11091,
 12778, 12831, 12845, 13214, 13618,
 13660, 13673, 13943, 14021, 14044,
 14206, 14246, 14678, 15406, 15555,
 15570, 15597, 15712, 17070, 17076,
 17102, 17114, 17122, 17154, 17161,
 17185, 17189, 17267, 17303, 17321,
 17674, 17706, 17761, 17776, 17784,
 18223, 18226, 18236, 18271, 18276,
 18277, 18453, 18638, 19341, 19356,
 19378, 19392, 20258, 20297, 20300,
 20463, 20565, 20593, 20604, 20642,
 20703, 21053, 21150, 21298, 21336,
 21346, 21650, 21721, 21902, 24268,

- 24289, 24301, 24311, 24406, 24462,
 24471, 24563, 24578, 24580, 24724,
 24822, 24832, 24844, 24857, 24858,
 24877, 24878, 24892, 24893, 24904,
 24905, 24972, 25021, 25056, 25059,
 25075, 25082, 25135, 25138, 25254,
 25255, 25256, 25257, 25260, 25356,
 25470, 25476, 25706, 25750, 25965,
 26036, 26297, 26315, 26365, 26375,
 26662, 26663, 26664, 26665, 26666,
 26667, 26895, 26907, 26908, 26937,
 26949, 26956, 26974, 27037, 27072,
 27086, 27132, 27139, 27218, 27230,
 27332, 27335, 27346, 27400, 27476,
 27548, 27551, 27558, 27591, 27592,
 27595, 27816, 28072, 28083, 28270,
 28280, 28338, 28439, 28527, 28576,
 28590, 28737, 28773, 28785, 28798,
 28801, 28804, 28807, 28832, 28934,
 28938, 28998, 29015, 29021, 29605,
 29675, 29736, 29737, 29739, 29740,
 29760, 29777, 29845, 29943, 30042,
 30096, 30170, 30240, 30245, 30351,
 30383, 30394, 30501, 31091, 31220,
 31273, 31279, 32455, 32704, 33316
 \if_mode_horizontal:
 30, 1397, 1398, 8700
 \if_mode_inner: . 30, 1397, 1400, 8702
 \if_mode_math: .. 30, 1397, 1397, 8704
 \if_mode_vertical:
 30, 1397, 1399, 2440, 8698
 \if_predicate:w
 65, 68, 74, 8366, 8366,
 8468, 8529, 8544, 8555, 8570, 8581
 \if_true: 29, 68, 1386,
 1386, 1885, 1891, 1901, 1907, 12224,
 12228, 13272, 13278, 20344, 20350
 \if_vbox:N ... 323, 36474, 36475, 36480
 \ifabsdim 936
 \ifabsnum 937
 \ifcase 261
 \ifcat 262
 \ifcondition 837
 \ifcsname 383, 682, 500
 \ifdbbox 1145
 \ifddir 1146
 \ifdefined 501
 \ifdim 263
 \ifeof 264
 \iffalse 265
 \IfFileExists 685
 \iffontchar 502
 \ifhbox 266
 \ifhmode 267
 \ifincsname 671
 \ifinner 268
 \ifjfont 1147
 \ifmbox 1148
 \ifmdir 1149
 \ifmmode 269
 \ifnum 10, 22, 52, 56, 270
 \ifodd 271
 \ifpdfabsdim 623
 \ifpdfabsnum 624
 \IfPDFManagementActiveTF .. 41141, 41142
 \ifpdfprimitive 625
 \ifprimitive 774
 \iftbox 1150
 \iftdir 1152
 \iftfont 1151
 \iftrue 272
 \ifvbox 273
 \ifvmode 274
 \ifvoid 275
 \ifx 4, 8, 13, 17, 53, 54, 61, 276
 \ifybox 1153
 \ifydir 1154
 \ignoreligaturesinfont 938
 \ignoreprimitiveerror 1214
 \ignorespaces 277
 \IJ 33341, 34882, 35764
 \ij 33341, 34882, 35776
 \immediate 68, 278
 \immediateassigned 838
 \immediateassignment 839
 in 287
 \indent 279
 inf 286
 \infty 25257, 25258
 inherit commands:
 .inherit:n 249, 23024
 \inhibitglue 1155
 \inhibitxspcode 1156
 \initcatcodetable 840
 initial commands:
 .initial:n 249, 23026
 \input 14, 280
 \inputlineno 281
 \insert 282
 \insertht 939
 \insertpenalties 283
 int commands:
 \int_abs:n 172,
 881, 16887, 18229, 18229, 24086, 42601
 \int_add:Nn 174, 4507,
 5675, 6497, 6498, 6755, 6827, 10955,
18360, 18360, 18368, 42157, 42540

- \int_case:nn [177](#), [898](#), [18524](#), [18539](#),
[18710](#), [18716](#), [31897](#), [34163](#), [34185](#),
[34190](#), [34202](#), [34638](#), [34653](#), [34720](#)
- \int_case:nnTF [177](#),
[4256](#), [8332](#), [18116](#), [18524](#), [18524](#),
[18529](#), [18534](#), [19692](#), [25166](#), [29989](#),
[36115](#), [36149](#), [36184](#), [36209](#), [36244](#)
- \int_compare:n [18431](#)
- \int_compare:nNn [18484](#)
- \int_compare:nNnTF
..... [175–178](#), [270](#), [893](#),
[3204](#), [4077](#), [4545](#), [4557](#), [4710](#), [5040](#),
[5042](#), [5905](#), [6705](#), [7057](#), [7216](#), [7616](#),
[7809](#), [8349](#), [8783](#), [8789](#), [9249](#), [10646](#),
[10767](#), [11345](#), [11355](#), [11710](#), [11749](#),
[12487](#), [12515](#), [12530](#), [12538](#), [12576](#),
[12583](#), [13392](#), [13399](#), [13468](#), [14074](#),
[14076](#), [14085](#), [14330](#), [14335](#), [14345](#),
[14348](#), [14402](#), [14872](#), [14948](#), [15754](#),
[16852](#), [16973](#), [17641](#), [17642](#), [17644](#),
[17646](#), [17719](#), [17902](#), [17909](#), [18311](#),
[18317](#), [18484](#), [18498](#), [18499](#), [18500](#),
[18501](#), [18502](#), [18503](#), [18504](#), [18505](#),
[18506](#), [18507](#), [18508](#), [18509](#), [18510](#),
[18511](#), [18512](#), [18513](#), [18514](#), [18515](#),
[18548](#), [18600](#), [18608](#), [18617](#), [18623](#),
[18635](#), [18706](#), [18795](#), [18801](#), [18807](#),
[18827](#), [18981](#), [19000](#), [19002](#), [19044](#),
[19766](#), [19768](#), [19773](#), [19782](#), [19803](#),
[19820](#), [19837](#), [20785](#), [20928](#), [20946](#),
[21922](#), [21971](#), [21974](#), [23797](#), [24024](#),
[24029](#), [24036](#), [24142](#), [24751](#), [26691](#),
[27848](#), [29972](#), [30120](#), [30122](#), [31141](#),
[31340](#), [31342](#), [31423](#), [31566](#), [31744](#),
[31777](#), [31919](#), [31925](#), [31936](#), [31962](#),
[31965](#), [32106](#), [32119](#), [32218](#), [32274](#),
[32282](#), [32290](#), [32303](#), [32358](#), [32361](#),
[32385](#), [32830](#), [32835](#), [32845](#), [32848](#),
[32866](#), [32875](#), [33840](#), [33879](#), [36168](#),
[36327](#), [39654](#), [39926](#), [39932](#), [40493](#),
[40968](#), [41202](#), [41263](#), [41283](#), [41296](#)
- \int_compare:nTF
[175](#), [176](#), [178](#), [271](#), [988](#), [6128](#), [6168](#),
[8068](#), [8297](#), [8298](#), [8303](#), [8305](#), [10071](#),
[10073](#), [10355](#), [10599](#), [18431](#), [18572](#),
[18580](#), [18589](#), [18595](#), [30180](#), [30804](#),
[30806](#), [31783](#), [36104](#), [36324](#), [36356](#)
- \int_compare_p:n [176](#), [6175](#), [18431](#)
- \int_compare_p:nNn . [29](#), [175](#), [3635](#),
[5750](#), [5751](#), [8797](#), [9084](#), [9153](#), [9155](#),
[9157](#), [10514](#), [11275](#), [11276](#), [11334](#),
[11335](#), [18484](#), [18492](#), [18494](#), [18496](#),
[31699](#), [33777](#), [34773](#), [34774](#), [34794](#),
[34795](#), [35088](#), [36137](#), [36138](#), [36229](#),
[36230](#), [36266](#), [36267](#), [36297](#), [41076](#),
[41077](#), [41084](#), [41087](#), [41088](#), [41096](#),
[41099](#), [41100](#), [41143](#), [41267](#), [41268](#)
- \int_const:Nn
..... [173](#), [4438](#), [4439](#), [4440](#), [4441](#),
[4861](#), [4862](#), [4863](#), [4864](#), [4865](#), [4866](#),
[4870](#), [4871](#), [4872](#), [4873](#), [4874](#), [4875](#),
[4876](#), [4877](#), [4878](#), [4879](#), [4880](#), [4881](#),
[4882](#), [8998](#), [9094](#), [9096](#), [9098](#), [9099](#),
[9100](#), [9140](#), [10264](#), [10443](#), [10509](#),
[10510](#), [14455](#), [14456](#), [18306](#), [18306](#),
[18308](#), [19010](#), [19011](#), [19012](#), [19013](#),
[19014](#), [19015](#), [19016](#), [19017](#), [19018](#),
[19019](#), [19020](#), [19021](#), [19022](#), [19023](#),
[19068](#), [19069](#), [19070](#), [20776](#), [24249](#),
[24250](#), [24251](#), [24252](#), [24253](#), [24254](#),
[24255](#), [24439](#), [24440](#), [24441](#), [24444](#),
[24445](#), [24446](#), [24449](#), [24450](#), [24451](#),
[24817](#), [25094](#), [25095](#), [25096](#), [25097](#),
[25098](#), [25099](#), [25100](#), [25101](#), [25102](#),
[25103](#), [25104](#), [25105](#), [25106](#), [25107](#),
[25108](#), [29008](#), [30302](#), [32127](#), [40777](#),
[40790](#), [40953](#), [41058](#), [42306](#), [42544](#)
- \int_decr:N [174](#),
[3365](#), [3366](#), [3367](#), [3430](#), [3431](#), [3440](#),
[3441](#), [3450](#), [3451](#), [3720](#), [7198](#), [7275](#),
[7498](#), [7593](#), [18372](#), [18374](#), [18381](#), [42160](#)
- \int_div_round:nn .. [172](#), [18261](#), [18282](#)
- \int_div_truncate:nn
..... [172](#), [173](#), [3635](#), [8812](#), [8833](#),
[9097](#), [14571](#), [14576](#), [15224](#), [15225](#),
[15281](#), [15463](#), [15629](#), [15640](#), [18261](#),
[18261](#), [18721](#), [18820](#), [18840](#), [20788](#),
[32022](#), [32035](#), [32040](#), [32052](#), [32140](#),
[32362](#), [32370](#), [32474](#), [41048](#), [42648](#)
- \int_do_until:nn
..... [178](#), [18570](#), [18592](#), [18596](#)
- \int_do_until:nNnn
..... [177](#), [18598](#), [18620](#), [18624](#)
- \int_do_while:nn
..... [178](#), [18570](#), [18586](#), [18590](#)
- \int_do_while:nNnn
..... [178](#), [18598](#), [18614](#), [18618](#)
- \int_eval:n [21](#), [36](#), [172–177](#),
[185](#), [382](#), [385](#), [413](#), [490](#), [646](#), [737](#),
[883](#), [901](#), [1052](#), [1053](#), [1057](#), [1058](#),
[1063](#), [1097](#), [1148](#), [1173](#), [1175](#), [2074](#),
[2103](#), [2119](#), [3306](#), [3571](#), [3572](#), [3843](#),
[3925](#), [3929](#), [3952](#), [5919](#), [6127](#), [7118](#),
[8072](#), [8117](#), [8118](#), [8338](#), [8653](#), [9119](#),
[10078](#), [10317](#), [10568](#), [10891](#), [10949](#),
[11325](#), [11326](#), [11349](#), [11359](#), [11366](#),
[11367](#), [12541](#), [12586](#), [13018](#), [13023](#),
[13031](#), [13385](#), [13393](#), [13401](#), [13430](#),

- 13434, 13443, 13450, 13485, 13495,
 14068, 14081, 14106, 14130, 14131,
 14143, 14148, 14179, 14196, 14235,
 14326, 14355, 14359, 14366, 14375,
 14526, 14546, 14564, 14841, 15367,
 15382, 15410, 15559, 15564, 15582,
 15726, 16837, 17637, 17895, 17903,
 17911, 18090, 18211, 18211, 18307,
 18527, 18532, 18537, 18542, 18702,
 18790, 18792, 18922, 18932, 18967,
 18978, 18984, 18995, 19026, 19063,
 19067, 19660, 19672, 19760, 19770,
 19784, 19791, 19807, 19869, 19871,
 19939, 19941, 19945, 19947, 19951,
 19953, 19957, 19959, 19993, 19994,
 21444, 21949, 21987, 21992, 22000,
 22006, 22015, 23796, 23882, 23930,
 23948, 23964, 24023, 24061, 24062,
 24113, 24130, 24259, 24980, 24987,
 30451, 30454, 30455, 30548, 30549,
 31100, 31136, 31184, 31246, 31254,
 31447, 31562, 31735, 32005, 32057,
 32060, 32065, 32071, 32084, 32099,
 32150, 32224, 32229, 32270, 32296,
 32336, 32395, 32407, 32438, 32449,
 32468, 32518, 32600, 32608, 32617,
 32621, 33826, 34244, 34273, 34307,
 34385, 34403, 34420, 34445, 36504,
 36514, 39934, 39977, 40023, 40976,
 40981, 40988, 40993, 41046, 41054,
 41280, 41944, 41946, 42445, 42600
 \int_eval:w ... 172, 384, 387, 3642,
 3910, 3921, 10842, 10851, 10876,
 10888, 14099, 14558, 18043, 18211,
 18213, 19108, 19143, 22021, 23978,
 24155, 24162, 24163, 24174, 27790
 \int_format:nn 182, 16836, 16836
 \int_from_alpha:n ... 182, 18965, 18965
 \int_from_base:nn
 183, 18982, 18982, 19005, 19007, 19009
 \int_from_bin:n
 182, 295, 1303, 19004, 19004, 31424
 \int_from_hex:n
 182, 19004, 19006, 39005, 39006, 39007
 \int_from_oct:n ... 182, 19004, 19008
 \int_from_roman:n .. 183, 19024, 19024
 \int_gadd:Nn 174, 18360,
 18364, 18369, 41308, 42239, 42541
 \int_gdecr:N 174,
 3986, 4116, 10478, 12964, 13962,
 17968, 18025, 18372, 18378, 18383,
 18700, 19594, 21409, 21887, 26879,
 35490, 35500, 41197, 41355, 42242
 \int_gincr:N 174, 3975, 4106, 6273,
 10469, 12955, 13951, 17960, 18019,
 18372, 18376, 18382, 18675, 18686,
 19585, 20781, 21404, 21866, 21873,
 23790, 24014, 26858, 26865, 31131,
 31655, 35484, 35494, 39641, 40234,
 40239, 40244, 41178, 41345, 42241
 .int_gset:N 249, 23036
 \int_gset:Nn
 174, 884, 2322, 6290, 9476, 18384,
 18386, 18389, 36513, 36515, 41188,
 41192, 41222, 41261, 41271, 41274,
 41289, 41305, 41311, 42243, 42539
 \int_gset_eq:NN
 173, 10673, 18352, 18354,
 18355, 18398, 18410, 41284, 42238
 \int_gset_regex_count:NNn
 174, 18390, 18405, 18413
 \int_gset_regex_count:Nnn
 174, 18390, 18393, 18401
 \int_gsub:Nn 174, 18360,
 18366, 18371, 31145, 42240, 42543
 \int_gzero:N
 ... 173, 2312, 6253, 6270, 18342,
 18343, 18345, 18349, 41296, 42237
 \int_gzero_new:N
 173, 18346, 18348, 18351, 41187
 \int_if_even:n 18562
 \int_if_even:nTF 177, 18554
 \int_if_even_p:n 177, 18554
 \int_if_exist:N 18356, 18358
 \int_if_exist:NTF 173,
 5599, 5654, 18347, 18349, 18356,
 19038, 19042, 40750, 40776, 40958
 \int_if_exist_p:N 173, 18356
 \int_if_odd:n 18554
 \int_if_odd:nTF 177,
 7415, 7438, 7512, 11101, 18554, 28191
 \int_if_odd_p:n 177, 6201, 18554
 \int_if_zero:n 18516
 \int_if_zero:nTF 177, 18516
 \int_if_zero_p:n 177, 18516
 \int_incr:N . 174, 3278, 3375, 3376,
 3761, 3803, 3816, 3834, 4372, 4373,
 5490, 6133, 6297, 6337, 6426, 6756,
 6852, 7186, 7258, 7492, 7497, 7532,
 7590, 7675, 7676, 7712, 7739, 7839,
 7840, 8002, 17744, 18372, 18372,
 18380, 22692, 23947, 24102, 24136,
 24187, 31044, 31234, 39967, 42159
 \int_log:N ... 183, 19064, 19064, 19065
 \int_log:n 183, 19066, 19066
 \int_max:nn 173,
 1266, 5978, 5979, 5986, 5987, 6284,
 6450, 7805, 7807, 16715, 16722,

- 16732, 16742, 16754, 17013, 17014,
18229, 18237, 28047, 29237, 42646
\int_min:nn 173, 1270, 18229, 18245,
32345, 41291, 41292, 41293, 42647
\int_mod:nn
... 173, 8814, 8835, 9095, 14877,
14941, 15225, 15226, 15464, 18261,
18284, 18711, 18811, 18831, 20790,
32054, 32373, 32487, 41055, 42649
\int_new:N 173, 3183, 3184,
3185, 3186, 3187, 3188, 3189, 3190,
3191, 3192, 3193, 3622, 3623, 3624,
3625, 4095, 4425, 4426, 4427, 4437,
4859, 4860, 4867, 4868, 4885, 6218,
6220, 6221, 6222, 6225, 6248, 6249,
6630, 6631, 6632, 6633, 6634, 6635,
6636, 6638, 6639, 6640, 6641, 6644,
6645, 6646, 6906, 7459, 7462, 7463,
7464, 7470, 7471, 8353, 8713, 10676,
10679, 10681, 10694, 14923, 18300,
18300, 18305, 18313, 18319, 18347,
18349, 19080, 19081, 19082, 19083,
19084, 19085, 20775, 22457, 23774,
23777, 23778, 24010, 31024, 31124,
31125, 31361, 31513, 38643, 39582,
40892, 41175, 41227, 41228, 41229,
41230, 41231, 41232, 41233, 41333
\int_rand:n
... 183, 23939, 24123, 30546, 30546
\int_rand:nn
79, 183, 1269, 1276, 13408, 17917,
19068, 19821, 19826, 30449, 30449
\int_range:nn 1270
.int_set:N 249, 23036
\int_set:Nn 174, 382, 2304,
2318, 2319, 2324, 2326, 3198, 3200,
3202, 3224, 3225, 3240, 3248, 3249,
3261, 3262, 3280, 3283, 3715, 3778,
4130, 4226, 4229, 4360, 5681, 6219,
6283, 6286, 6324, 6326, 6395, 6446,
6447, 6457, 6468, 6492, 6510, 6559,
6691, 6693, 6717, 6760, 6761, 6802,
6837, 7537, 7609, 7611, 7755, 7781,
7804, 7806, 10429, 10431, 10652,
10654, 10677, 10687, 10700, 10747,
10753, 10765, 10770, 12488, 12523,
12643, 12644, 15002, 15051, 15104,
17745, 18384, 18384, 18388, 22526,
22698, 23159, 24178, 31623, 31624,
31639, 31810, 31825, 31859, 36523,
36524, 36525, 36526, 42161, 42538
\int_set_eq:NN 173, 3241, 3271, 4498,
4968, 4972, 4981, 4983, 5026, 5093,
5389, 5489, 5502, 5601, 6239, 6259,
6276, 6281, 6301, 6335, 6336, 6386,
6489, 6490, 6542, 6591, 6669, 6692,
6696, 6697, 6711, 6715, 6718, 6757,
6767, 6898, 6899, 7488, 7705, 7998,
8979, 11152, 12486, 12489, 18352,
18352, 18353, 18396, 18408, 42156
\int_set_gregex_count:NNn 18390
\int_set_regex_count:NNn
..... 174, 18390, 18402
\int_set_regex_count:Nnn
.... 174, 18390, 18390, 18392, 18404
\int_show:N .. 183, 19060, 19060, 19061
\int_show:n 183, 646, 903, 19062, 19062
\int_sign:n
172, 992, 18215, 18215, 31706, 42602
\int_step_function:nN
179, 18626, 18662, 18666, 31828, 31829
\int_step_function:nnN . 179, 6768,
7605, 18626, 18664, 18667, 20067,
31830, 31831, 31832, 31833, 31853
\int_step_function:nnnN
.. 75, 179, 894, 1151, 7784, 7792,
18626, 18626, 18663, 18665, 18668,
18699, 42710, 42714, 42718, 42722
\int_step_inline:nn 179,
1058, 17732, 18669, 18669, 24017,
31543, 31577, 31634, 32379, 32425
\int_step_inline:nnn .. 179, 3332,
6684, 8356, 9191, 9193, 9195, 10268,
10521, 14861, 14870, 18669, 18671,
31550, 31553, 31811, 32292, 32397
\int_step_inline:nnnn
179, 1153, 18669, 18670, 18672, 18673
\int_step_tokens:nn 179, 18666, 18666
\int_step_tokens:nnn 179, 18666, 18667
\int_step_tokens:nnnn
..... 179, 18666, 18668
\int_step_variable:nNn
..... 180, 18669, 18680
\int_step_variable:nnNn
..... 180, 18669, 18682
\int_step_variable:nnnnN
.... 180, 18669, 18681, 18683, 18684
\int_sub:Nn 174, 4502,
5204, 6545, 6553, 6562, 9420, 10963,
18360, 18362, 18370, 42158, 42542
\int_to_Alph:n 180, 182, 18725, 18757
\int_to_alph:n 180–182, 18725, 18725
\int_to_arabic:n
..... 180, 18702, 18702, 18703
\int_to_Base:n 181
\int_to_base:n 181
\int_to_Base:nn
..... 181, 183, 18789, 18791, 18916

```

\int_to_base:nn ..... 181,
183, 18789, 18789, 18912, 18914, 18918
\int_to_bin:n .....
..... 181, 182, 16872, 18911, 18911
\int_to_Hex:n .... 181, 182, 4713,
16874, 18911, 18915, 32527, 39576
\int_to_hex:n . 181, 182, 18911, 18913
\int_to_oct:n .....
..... 181, 182, 16873, 18911, 18917
\int_to_Roman:n 182, 183, 18919, 18929
\int_to_roman:n 182, 183, 18919, 18919
\int_to_symbols:nnn ..... 180,
181, 18704, 18704, 18720, 18727, 18759
\int_until_do:nn .....
..... 178, 18570, 18578, 18583
\int_until_do:nNnn .....
..... 178, 18598, 18606, 18611
\int_use:N ..... 171,
175, 1090, 1096, 2323, 2325, 2327,
3977, 4108, 4128, 5000, 5087, 5165,
5176, 5185, 5189, 5200, 5201, 5207,
5208, 5214, 5215, 5372, 6201, 6291,
6296, 6317, 6319, 6424, 6437, 6438,
6838, 6890, 6988, 6999, 7154, 7537,
7621, 7622, 7813, 7814, 8350, 8823,
8825, 8828, 8844, 8846, 8850, 8853,
8858, 9380, 10080, 10432, 10471,
10648, 12957, 12959, 13953, 13957,
15363, 15387, 15401, 15421, 15428,
15610, 15719, 15724, 15740, 17961,
17967, 18021, 18023, 18414, 18414,
18415, 18678, 18689, 19587, 19589,
21403, 21411, 21869, 21876, 22526,
22699, 23159, 23791, 23885, 23933,
26861, 26868, 31055, 31657, 31659,
31690, 31739, 35486, 35488, 35496,
35498, 39647, 40960, 41180, 41195,
41217, 42442, 42443, 42444, 42479
\int_value:w .....
185, 387, 464, 609, 881, 887, 987,
1052, 1053, 1057, 1058, 1067, 1073,
1077, 1090, 1098, 1105, 1108, 1113,
1120, 1149, 1150, 1159, 1167, 1175,
1242, 1247, 1261, 1807, 3306, 3642,
3656, 3861, 3908, 3910, 3921, 3929,
3948, 3950, 3958, 4170, 4205, 4269,
4289, 4298, 4706, 5233, 5239, 5269,
5271, 5280, 5281, 5396, 5881, 5896,
6923, 6924, 6935, 7646, 8504, 8507,
8653, 9136, 10842, 10851, 13443,
13450, 14068, 14069, 14081, 14099,
14106, 14129, 14130, 14131, 14143,
14179, 14794, 14918, 15281, 15359,
15367, 15382, 15410, 17667, 17677,
18031, 18043, 18199, 18199, 18217,
18218, 18231, 18232, 18239, 18240,
18241, 18247, 18248, 18249, 18263,
18265, 18266, 18283, 18286, 18287,
18288, 18295, 18434, 18438, 18468,
18629, 18630, 18631, 18660, 18875,
18908, 19108, 19143, 19993, 19994,
20094, 21678, 21893, 21927, 21934,
21957, 21965, 21966, 22021, 24033,
24036, 24061, 24062, 24108, 24113,
24155, 24162, 24163, 24174, 24333,
24334, 24335, 24336, 24337, 24351,
24500, 24562, 24580, 24891, 25024,
25038, 25040, 25042, 25045, 25081,
25219, 25249, 25250, 25287, 25295,
25426, 25431, 25433, 25442, 25446,
25484, 25492, 25495, 25501, 25512,
25523, 25529, 25530, 25533, 25576,
25586, 25588, 25604, 25606, 25629,
25643, 25719, 25720, 25794, 25882,
26661, 26694, 27047, 27048, 27049,
27051, 27097, 27100, 27103, 27126,
27128, 27149, 27151, 27160, 27162,
27166, 27187, 27194, 27200, 27210,
27212, 27226, 27234, 27242, 27287,
27289, 27306, 27308, 27311, 27314,
27369, 27377, 27379, 27381, 27383,
27386, 27389, 27391, 27411, 27413,
27417, 27424, 27426, 27430, 27453,
27456, 27464, 27466, 27469, 27470,
27471, 27472, 27487, 27490, 27493,
27496, 27505, 27508, 27511, 27514,
27521, 27523, 27530, 27539, 27541,
27543, 27569, 27571, 27580, 27582,
27586, 27603, 27624, 27628, 27640,
27643, 27646, 27649, 27652, 27655,
27658, 27661, 27665, 27677, 27681,
27685, 27688, 27709, 27711, 27713,
27723, 27747, 27750, 27762, 27764,
27770, 27773, 27790, 27810, 27867,
27872, 27874, 27882, 27885, 27888,
27891, 27894, 27897, 27906, 27918,
27926, 27928, 27938, 27940, 27947,
27957, 27959, 27962, 27965, 27968,
27971, 27984, 27986, 27995, 27997,
28005, 28007, 28017, 28020, 28023,
28030, 28045, 28064, 28067, 28125,
28139, 28141, 28148, 28161, 28163,
28165, 28190, 28206, 28213, 28214,
28260, 28262, 28263, 28264, 28305,
28307, 28353, 28360, 28367, 28388,
28390, 28392, 28394, 28407, 28411,
28412, 28413, 28414, 28415, 28420,
28426, 28428, 28435, 28453, 28454,

```

- 28455, 28456, 28457, 28458, 28465,
28467, 28469, 28471, 28473, 28478,
28480, 28482, 28484, 28486, 28488,
28514, 28523, 28539, 28544, 28548,
28607, 28656, 28724, 28733, 28741,
28752, 28754, 28757, 28760, 28850,
28887, 28889, 28892, 28895, 28898,
28901, 28908, 28911, 28913, 28917,
28939, 28941, 28974, 29044, 29054,
29059, 29069, 29211, 29243, 29252,
29484, 29485, 29496, 29499, 29502,
29505, 29508, 29511, 29514, 29517,
29520, 29538, 29548, 29557, 29575,
29584, 29591, 29601, 29662, 29671,
29716, 29759, 29776, 29832, 29844,
29855, 30068, 30144, 30191, 30236,
30244, 30246, 30248, 30314, 30337,
30391, 30431, 30443, 30454, 30455,
30485, 30488, 30491, 30493, 30495,
30502, 30505, 30513, 30520, 30525,
31100, 31184, 31246, 31254, 31267,
31268, 31269, 31279, 32746, 41905
\int_while_do:nn
..... 178, 18570, 18570, 18575
\int_while_do:nNnn
..... 178, 18598, 18598, 18603
\int_zero:N
.... 173, 3716, 3717, 3718, 3817,
4980, 5202, 5638, 6165, 6238, 6269,
6690, 6967, 7482, 7483, 7531, 7718,
7993, 10807, 17726, 18342, 18342,
18344, 18347, 22689, 23944, 24099,
24128, 31041, 31231, 39964, 42155
\int_zero_new:N
..... 173, 18346, 18346, 18350
\c_max_char_int
..... 184, 4710, 19070, 20008, 20776
\c_max_int
. 184, 262, 554, 1269, 1270, 4136,
4137, 4138, 19069, 30496, 36501,
36507, 38588, 38591, 41268, 41278
\c_max_register_int
..... 184, 449, 1439, 3200,
3225, 3262, 10078, 10080, 17719, 18199
\c_one_int .. 184, 3818, 4140, 4787,
4910, 5923, 5979, 5987, 6030, 6175,
6281, 6396, 6469, 6480, 6491, 6494,
6511, 6551, 6560, 6685, 6688, 6696,
6717, 6770, 6787, 6788, 6798, 6860,
6861, 6935, 7105, 7118, 7140, 7606,
7787, 7795, 8358, 18373, 18375,
18377, 18379, 18663, 18665, 19068,
19108, 19143, 20704, 24155, 24174,
27675, 27679, 27683, 27737, 28354,
28506, 28648, 28958, 29825, 29910,
30354, 30358, 30365, 30553, 31094
\g_tmpa_int 184, 19080
\l_tmpa_int 4, 55, 184, 19080
\g_tmpb_int 184, 19080
\l_tmpb_int 4, 184, 19080
\c_zero_int 184, 393, 405,
737, 1438, 1805, 1807, 3841, 3865,
3922, 4030, 4142, 4403, 4545, 4557,
4998, 5496, 5751, 5892, 5978, 5986,
6168, 6267, 6289, 6380, 6406, 6467,
6507, 6558, 6620, 6951, 6958, 6986,
7057, 7125, 7196, 7205, 7216, 7248,
7257, 7274, 7297, 7567, 7583, 7617,
7650, 7658, 7709, 7711, 7783, 7805,
7807, 7810, 8979, 11152, 11476,
12486, 13448, 13497, 13821, 13826,
14150, 14185, 14916, 15716, 18296,
18297, 18311, 18342, 18343, 18418,
18426, 18518, 18635, 19068, 20007,
20701, 20702, 20703, 21744, 21922,
24018, 24142, 24581, 24823, 24827,
24829, 24833, 24837, 24850, 24863,
24870, 24883, 24895, 24907, 24916,
24919, 24930, 25039, 25044, 26676,
26708, 27701, 27736, 27738, 27739,
27740, 28541, 28608, 28833, 28851,
28875, 28907, 29781, 30145, 30337,
30366, 30466, 30533, 31067, 31369
int internal commands:
__int_abs:N 18229, 18231, 18235
__int_case:nnTF 18524,
18527, 18532, 18537, 18542, 18544
__int_case:nw
..... 18524, 18545, 18546, 18550
__int_case_end:nw 18524, 18549, 18552
__int_compare:nnN
. 888, 18431, 18463, 18471, 18473,
18475, 18477, 18479, 18481, 18483
__int_compare:NNw
..... 888, 18431, 18443, 18447
__int_compare:Nw
887, 889, 18431, 18439, 18441, 18468
__int_compare:w
..... 888, 18431, 18433, 18436
__int_compare_!=:NNw 18431
__int_compare_<:NNw 18431
__int_compare_<=:NNw 18431
__int_compare_=:NNw 18431
__int_compare_==:NNw 18431
__int_compare_>:NNw 18431
__int_compare_>=:NNw 18431
__int_compare_end=:NNw . 888, 18431

__int_compare_error: 887,
 888, [18416](#), 18416, 18420, 18434, 18436
 __int_compare_error:Nw
 887–889, [18416](#), 18422, 18456
 __int_const:nN
 [18306](#), 18307, 18309, 18329
 __int_constdef:Nw . [18306](#), 18324,
 18335, 18336, 18337, 18339, 18340
 __int_div_truncate:NwNw
 [18261](#), 18264, 18269, 18292
 __int_eval:w . 382, 881, 882, 888,
 893, [18199](#), 18200, 18212, 18213,
 18218, 18232, 18240, 18241, 18248,
 18249, 18263, 18265, 18266, 18283,
 18286, 18287, 18288, 18295, 18327,
 18361, 18363, 18365, 18367, 18385,
 18387, 18434, 18468, 18486, 18518,
 18556, 18564, 18629, 18630, 18631,
 18660, 18848, 18875, 18881, 18908,
 32647, 42546, 42604, 42651, 42675,
 42691, 42692, 42713, 42717, 42721
 __int_eval_end:
 [18199](#), 18201, 18212, 18218,
 18232, 18267, 18283, 18289, 18298,
 18327, 18361, 18363, 18365, 18367,
 18385, 18387, 18486, 18556, 18564,
 18848, 18875, 18881, 18908, 32647
 __int_from_alpha:N
 900, [18965](#), 18978, 18980
 __int_from_alpha:nN
 900, [18965](#), 18970, 18974, 18977
 __int_from_base:N
 901, [18982](#), 18995, 18998
 __int_from_base:nnN
 901, [18982](#), 18987, 18991, 18994
 __int_from_roman:NN
 .. [19024](#), 19030, 19035, 19051, 19055
 \c__int_from_roman_C_int [19010](#)
 \c__int_from_roman_c_int [19010](#)
 \c__int_from_roman_D_int [19010](#)
 \c__int_from_roman_d_int [19010](#)
 __int_from_roman_error:w
 [19024](#), 19039, 19043, 19058
 \c__int_from_roman_I_int [19010](#)
 \c__int_from_roman_i_int [19010](#)
 \c__int_from_roman_L_int [19010](#)
 \c__int_from_roman_l_int [19010](#)
 \c__int_from_roman_M_int [19010](#)
 \c__int_from_roman_m_int [19010](#)
 \c__int_from_roman_V_int [19010](#)
 \c__int_from_roman_v_int [19010](#)
 \c__int_from_roman_X_int [19010](#)
 \c__int_from_roman_x_int [19010](#)
 __int_if_recursion_tail_stop:N .
 [18209](#), 18210, 19037
 __int_if_recursion_tail_stop_-
 do:Nn
 .. [18209](#), 18209, 18976, 18993, 19040
 \c__int_max_constdef_int [18306](#)
 __int_maxmin:wwN
 [18229](#), 18239, 18247, 18253
 __int_mod:ww ... [18261](#), 18286, 18291
 __int_pass_signs:wn
 900, [18955](#), 18955, 18958, 18969, 18986
 __int_pass_signs_end:wn
 [18955](#), 18960, 18964
 __int_sep: .. [18214](#), 18214, 18218,
 18221, 18240, 18241, 18248, 18249,
 18253, 18265, 18266, 18269, 18287,
 18288, 18291, 18292, 18629, 18630,
 18631, 18633, 18649, 18652, 18660
 __int_show:nN [19060](#)
 __int_sign:Nw .. [18215](#), 18217, 18221
 __int_step:NNnnnn
 [18669](#), 18676, 18687, 18696
 __int_step:Nw
 .. [18626](#), 18636, 18647, 18652, 18658
 __int_step:Nwnnn 893
 __int_step:w ... [18626](#), 18628, 18633
 __int_step:wwn 893
 \l__int_tmpa_int ... 18396, 18397,
 18398, 18408, 18409, 18410, [19084](#)
 \l__int_tmpb_int [19084](#)
 __int_to_Base:nn [18789](#), 18792, 18799
 __int_to_base:nn [18789](#), 18790, 18793
 __int_to_Base:nnN
 .. [18789](#), 18802, 18803, 18825, 18839
 __int_to_base:nnN
 .. [18789](#), 18796, 18797, 18805, 18819
 __int_to_Base:nnnN
 [18789](#), 18830, 18837
 __int_to_base:nnnN
 [18789](#), 18810, 18817
 __int_to_Letter:n
 [18789](#), 18828, 18831, 18878
 __int_to_letter:n
 [18789](#), 18808, 18811, 18845
 __int_to_roman:N
 [18919](#), 18921, 18924, 18927
 __int_to_roman:w 888, 899,
 1419, 1420, [18199](#), 18444, 18922, 18932
 __int_to_Roman_aux:N
 18931, 18934, 18937
 __int_to_Roman_c:w ... [18919](#), 18951
 __int_to_roman_c:w ... [18919](#), 18943
 __int_to_Roman_d:w ... [18919](#), 18952
 __int_to_roman_d:w ... [18919](#), 18944

- __int_to_Roman_i:w ... [18919](#), [18947](#)
- __int_to_roman_i:w ... [18919](#), [18939](#)
- __int_to_Roman_l:w ... [18919](#), [18950](#)
- __int_to_roman_l:w ... [18919](#), [18942](#)
- __int_to_Roman_m:w ... [18919](#), [18953](#)
- __int_to_roman_m:w ... [18919](#), [18945](#)
- __int_to_Roman_Q:w ... [18919](#), [18954](#)
- __int_to_roman_Q:w ... [18919](#), [18946](#)
- __int_to_Roman_v:w ... [18919](#), [18948](#)
- __int_to_roman_v:w ... [18919](#), [18940](#)
- __int_to_Roman_x:w ... [18919](#), [18949](#)
- __int_to_roman_x:w ... [18919](#), [18941](#)
- __int_to_symbols:nnnn
..... [18704](#), [18708](#), [18718](#), [18724](#)
- __int_use_none_delimit_by_s-
stop:w [18206](#), [18206](#), [18466](#)
- intarray commands:
- \intarray_const_from_clist:Nn ...
... [262](#), [23941](#), [23941](#), [23950](#), [24125](#),
[24125](#), [24133](#), [28662](#), [29267](#), [42307](#)
- \intarray_count:N
... [263](#), [384](#), [14878](#), [14941](#), [23797](#),
[23800](#), [23841](#), [23855](#), [23885](#), [23933](#),
[23939](#), [24024](#), [24027](#), [24029](#), [24030](#),
[24033](#), [24033](#), [24034](#), [24042](#), [24052](#),
[24100](#), [24123](#), [24142](#), [24205](#), [31156](#)
- \intarray_gset:Nnn [263](#), [384](#),
[1057](#), [1059](#), [14862](#), [14874](#), [14887](#),
[14893](#), [23877](#), [23880](#), [23888](#), [24055](#),
[24057](#), [24064](#), [31538](#), [32445](#), [41029](#)
- \intarray_gzero:N
[262](#), [23890](#), [23899](#), [24097](#), [24097](#), [24106](#)
- \intarray_if_exist:N .. [24191](#), [24193](#)
- \intarray_if_exist:NTF
..... [263](#), [24191](#), [41027](#)
- \intarray_if_exist_p:N ... [263](#), [24191](#)
- \intarray_item:Nn [263](#), [384](#),
[1054](#), [1057](#), [1059](#), [14872](#), [14877](#),
[14911](#), [14940](#), [14948](#), [23901](#), [23928](#),
[23937](#), [23939](#), [24107](#), [24109](#), [24115](#),
[24123](#), [31745](#), [32472](#), [32486](#), [41040](#)
- \intarray_log:N
..... [263](#), [24195](#), [24197](#), [24198](#)
- \intarray_new:Nn
[262](#), [1050](#), [1056](#), [1059](#), [6647](#), [6648](#),
[7465](#), [7466](#), [7467](#), [7468](#), [7469](#), [14860](#),
[14869](#), [23786](#), [23793](#), [23803](#), [24011](#),
[24020](#), [24032](#), [31148](#), [31149](#), [31150](#),
[31534](#), [32139](#), [32423](#), [41028](#), [42349](#)
- \intarray_rand_item:N [263](#), [23938](#),
[23938](#), [23940](#), [24122](#), [24122](#), [24124](#)
- \intarray_show:N
[263](#), [1054](#), [1059](#), [24195](#), [24195](#), [24196](#)
- intarray internal commands:
- __intarray:w [23780](#), [23791](#)
- \l__intarray_bad_index_int
..... [23777](#), [23885](#), [23933](#)
- __intarray_bounds:NNnTF
..... [24037](#), [24037](#), [24067](#), [24118](#)
- __intarray_bounds_error:NNnw ...
..... [24037](#), [24040](#), [24043](#), [24048](#)
- __intarray_const_from_clist:nN .
..... [24125](#), [24130](#), [24134](#)
- __intarray_count:w [24007](#),
[24008](#), [24023](#), [24033](#), [24131](#), [24150](#)
- __intarray_entry:w
.. [24007](#), [24007](#), [24056](#), [24103](#), [24108](#)
- \g__intarray_font_int
..... [24010](#), [24014](#), [24016](#)
- __intarray_gset:Nnn [24055](#)
- __intarray_gset:Nnw .. [24059](#), [24065](#)
- __intarray_gset:w [23857](#), [23879](#)
- __intarray_gset:wTF .. [23857](#), [23882](#)
- __intarray_gset_count:Nw
..... [1049](#), [23775](#), [23796](#), [23841](#)
- __intarray_gset_overflow:Nnn . [24055](#)
- __intarray_gset_overflow:NNnn ..
..... [24079](#), [24087](#), [24091](#)
- __intarray_gset_overflow_-
test:nw [1055](#), [1059](#), [24001](#),
[24002](#), [24069](#), [24076](#), [24084](#), [24137](#)
- __intarray_gset_range:nNw ... [23975](#)
- __intarray_gset_range:Nw
..... [24176](#), [24179](#), [24181](#), [24188](#)
- __intarray_gset_range:w [23978](#)
- __intarray_item:Nw
..... [24107](#), [24111](#), [24116](#)
- __intarray_item:w [23901](#), [23927](#)
- __intarray_item:wTF .. [23901](#), [23930](#)
- \l__intarray_loop_int
.... [23774](#), [23944](#), [23947](#), [23948](#),
[24099](#), [24102](#), [24103](#), [24128](#), [24131](#),
[24136](#), [24138](#), [24178](#), [24186](#), [24187](#)
- __intarray_new:N
..... [23786](#), [23787](#), [23795](#),
[23943](#), [24011](#), [24011](#), [24022](#), [24127](#)
- __intarray_range_to_clist:w ...
..... [23960](#), [23963](#)
- __intarray_range_to_clist:ww ...
..... [24157](#), [24161](#), [24167](#), [24173](#)
- __intarray_sep:
.... [23773](#), [23773](#), [24061](#), [24062](#),
[24065](#), [24113](#), [24116](#), [24145](#), [24148](#),
[24155](#), [24162](#), [24163](#), [24167](#), [24174](#)
- __intarray_show:NN
..... [24195](#), [24197](#), [24199](#)

- __intarray_signed_max_dim:n ...
..... 24035, 24035, 24094, 24095
- \c__intarray_sp_dim
..... 24009, 24016, 24056
- __intarray_table 23815
- \g__intarray_table_int
..... 23777, 23790, 23791
- __intarray_to_clist:Nn
... 1054, 23951, 24140, 24140, 24206
- __intarray_to_clist:w
.. 23951, 24140, 24145, 24148, 24154
- \interactionmode 503
- \interlinepenalties 504
- \interlinepenalty 284
- interpolate 339
- ior commands:
 - \ior_close:N 94, 95,
10312, 10353, 10353, 10364, 11731,
32212, 32536, 32570, 40551, 40779
 - \ior_get:NN
95-97, 99, 10410, 10410, 10414, 10490
 - \ior_get:NNTF 96, 10410, 10411
 - \ior_get_term:nN 99, 10444, 10444
 - \ior_if_eof:N 663, 10394
 - \ior_if_eof:NNTF 98, 10394, 10416,
10436, 10476, 10495, 40541, 40768
 - \ior_if_eof_p:N 98, 10394
 - \ior_log:N ... 95, 10365, 10367, 10368
 - \ior_log_list: 95, 10381, 10382
 - \ior_map_break: 98, 10459, 10459,
10460, 10462, 10477, 10484, 10496,
10502, 32456, 32566, 40583, 40791
 - \ior_map_break:n 98, 10459, 10461
 - \ior_map_inline:Nn
..... 97, 10463, 10463, 11729
 - \ior_map_variable:NNn
..... 97, 10489, 10489, 32453
 - \ior_new:N 94, 10281, 10281, 10282,
10283, 10284, 11299, 32128, 40430
 - \ior_open:Nn 94, 693,
10285, 10285, 10287, 10289, 10298,
32203, 32451, 32491, 32537, 40519
 - \ior_open:NnTF 94, 10286, 10289
 - \ior_shell_open:Nn 94,
385, 10331, 10331, 11718, 40513, 40766
 - \ior_show:N .. 95, 10365, 10365, 10366
 - \ior_show_list: 95, 10381, 10381
 - \ior_str_get:NN
95, 96, 99, 10423, 10423, 10434, 10492
 - \ior_str_get:NNTF ... 96, 10423, 10424
 - \ior_str_get_term:nN 99, 10444, 10446
 - \ior_str_map_inline:Nn
..... 97, 10463, 10465,
32204, 32530, 32559, 40544, 40771
 - \ior_str_map_variable:NNn
..... 97, 10489, 10491
 - \g_tmpa_ior 102, 10283
 - \g_tmpb_ior 102, 10283
 - ior internal commands:
 - \l__ior_file_name_tl
..... 10288, 10291, 10293
 - __ior_get:NN
.. 10410, 10412, 10419, 10445, 10464
 - __ior_get_term:NnN
..... 10444, 10445, 10447, 10448
 - __ior_list:N
..... 10381, 10381, 10382, 10383
 - __ior_map_inline:NNn
..... 10463, 10464, 10466, 10467
 - __ior_map_inline:NNNn
..... 10463, 10470, 10473
 - __ior_map_inline_loop:NNN
..... 10463, 10476, 10480, 10487
 - __ior_map_variable:NNNn
..... 10489, 10490, 10492, 10493
 - __ior_map_variable_loop:NNNn ...
..... 10489, 10495, 10498, 10505
 - __ior_new:N
659, 10299, 10299, 10303, 10304, 10316
 - __ior_new_aux:N 10303, 10307
 - __ior_open_stream:Nn
..... 10310, 10314, 10318, 10322
 - __ior_shell_open:nN
..... 10331, 10334, 10337, 10346
 - __ior_show:NN
..... 10365, 10365, 10367, 10369
 - __ior_str_get:NN
.. 10423, 10425, 10439, 10447, 10466
 - \l__ior_stream_tl
..... 10266, 10313, 10317, 10324
 - \g__ior_streams_prop
660, 10267, 10325, 10358, 10373, 10388
 - \g__ior_streams_seq
..... 10265, 10313, 10359, 10360
 - \c__ior_term_ior 10264,
10281, 10355, 10361, 10397, 10454
 - \c__ior_term_noprompt_ior
..... 10443, 10453
 - \l__ior_tmp_tl
.. 10263, 10373, 10376, 10482, 10486
 - iow commands:
 - \iow_char:N 86, 100, 3559,
3562, 3563, 3587, 3588, 3595, 3596,
4684, 4685, 4692, 4694, 4696, 4698,
4700, 4702, 5346, 5347, 6081, 6088,
6089, 6090, 6214, 8040, 8043, 8044,
8049, 8083, 8092, 8096, 8101, 8121,
8123, 8124, 8126, 8129, 8131, 8136,

8138, 8140, 8145, 8149, 8152, 8153,
 8156, 8158, 8162, 8164, 8170, 8172,
 8176, 8178, 8182, 8187, 8189, 8231,
 8233, 8238, 8240, 8246, 8251, 8256,
 8260, 8270, 8273, 8277, 8278, 8282,
 8290, 8361, 9919, 9922, 9923, 9955,
 9990, 10148, 10675, 10675, 11788,
 11790, 11791, 11792, 14980, 28770,
 31876, 31878, 31879, 31882, 31884,
 31885, 31888, 31890, 31891, 31892,
 31896, 31903, 41873, 42745, 42746
 \iow_close:N
 .. 94, 95, 10563, 10597, 10597, 10608
 \iow_indent:n 101,
 672, 673, 8317, 9882, 10024, 10095,
 10103, 10119, 10725, 10725, 10728,
 10740, 10757, 10762, 14978, 15304,
 15492, 24767, 24779, 40276, 40305,
 40327, 40350, 40359, 40368, 40389
 \l_iow_line_count_int
 101, 102, 479, 673, 4079,
 4083, 9420, 10676, 10766, 10771, 10809
 \iow_log:N ... 95, 10609, 10611, 10612
 \iow_log:n 99, 382, 9620, 9627, 10664,
 10664, 10665, 10666, 13549, 41881
 \iow_log_list: 95, 10625, 10626
 \iow_new:N
 94, 10538, 10538, 10539, 10540, 10541
 \iow_newline: 86, 99–
 101, 385, 636, 669, 751, 3999, 4014,
 4029, 6136, 9442, 10674, 10674,
 10754, 10763, 10769, 11647, 26504,
 26505, 38575, 38576, 38577, 40700
 \iow_now:Nn 99, 100, 9026,
 10657, 10657, 10662, 10663, 10664,
 10665, 10667, 10668, 10669, 10670
 \iow_open:Nn . 94, 10554, 10554, 10560
 \iow_shell_open:Nn
 94, 385, 10581, 10581
 \iow_shipout:Nn 99, 100,
 669, 9058, 10640, 10640, 10642, 10643
 \iow_shipout_e:Nn 99,
 100, 10637, 10637, 10639, 41434, 41435
 \iow_shipout_x:Nn
 669, 41434, 41435, 41436
 \iow_show:N .. 95, 10609, 10609, 10610
 \iow_show:n .. 99, 10664, 10667, 10668
 \iow_show_list: 95, 10625, 10625
 \iow_term:n .. 99, 8351, 9454, 9610,
 9615, 9633, 9659, 10664, 10669,
 10670, 10671, 41327, 41343, 41358
 \iow_wrap:nnnN
 99–102, 646, 673, 751,
 9418, 9421, 9433, 9591, 9625, 9631,
 9638, 10717, 10723, 10728, 10740,
 10743, 10743, 10777, 13533, 13549
 \iow_wrap_allow_break: 101, 10714,
 10714, 10717, 10723, 10756, 10761
 \iow_wrap_allow_break:n 672
 \c_log_iow
 102, 664, 10509, 10599, 10664, 10665
 \c_term_iow
 102, 664, 665, 10509, 10538,
 10599, 10605, 10669, 10670, 10673
 \g_tmpa_iow 102, 10540
 \g_tmpb_iow 102, 10540
 iow internal commands:
 \l__iow_file_name_tl
 10553, 10556, 10558
 __iow_indent:n
 672, 10725, 10731, 10757
 __iow_indent_error:n
 672, 10725, 10737, 10762
 \l__iow_indent_int 10693,
 10807, 10826, 10938, 10955, 10963
 \l__iow_indent_tl .. 10693, 10808,
 10825, 10937, 10956, 10964, 10965
 \l__iow_line_break_bool
 10697, 10803, 10932, 10946, 10954,
 10962, 10970, 10972, 10977, 10979
 \l__iow_line_part_tl
 675, 677, 678, 10695, 10805,
 10818, 10839, 10897, 10900, 10931,
 10945, 10947, 10953, 10961, 10984
 \l__iow_line_target_int
 679, 10679, 10765,
 10767, 10770, 10933, 10938, 10973
 \l__iow_line_tl 10695, 10804, 10822,
 10912, 10928, 10944, 10945, 10953,
 10961, 10983, 10984, 10989, 10991
 __iow_list:N
 10625, 10625, 10626, 10627
 __iow_new:N
 .. 10542, 10542, 10546, 10547, 10567
 __iow_new_aux:N 10546, 10550
 \l__iow_newline_tl 10678,
 10763, 10764, 10766, 10769, 10988
 \l__iow_one_indent_int
 10680, 10955, 10963
 \l__iow_one_indent_tl
 671, 10680, 10956
 __iow_open_stream:Nn
 .. 10554, 10565, 10569, 10573, 10580
 __iow_sep: 675,
 10811, 10811, 10826, 10828, 10844,
 10851, 10853, 10876, 10888, 10938
 __iow_set_indent:n
 671, 10680, 10683, 10692

- __iow_shell_open:nN 10581, 10584, 10587, 10596
 - __iow_show:NN 10609, 10609, 10611, 10613
 - \l_iow_stream_tl 10519, 10564, 10568, 10575
 - \g__iow_streams_prop 668, 10520, 10576, 10602, 10617, 10632
 - \g__iow_streams_seq 10518, 10564, 10603, 10604
 - __iow_tmp:w 677, 10812, 10836, 10893, 10925, 10993, 11001
 - \l_iow_tmp_tl .. 10508, 10617, 10620
 - __iow_unindent:w 671, 10680, 10682, 10690, 10695
 - __iow_use_i_delimit_by_s-
stop:nw 10536, 10536, 10796
 - __iow_with:nNnn 10644, 10648, 10650, 10656
 - __iow_wrap_allow_break: 672, 10714, 10719, 10756
 - __iow_wrap_allow_break:n 10942, 10942
 - __iow_wrap_allow_break_error: .. 672, 10714, 10720, 10761
 - \c__iow_wrap_allow_break_marker-
tl 10699, 10719
 - __iow_wrap_break:w 10879, 10893, 10895
 - __iow_wrap_break_end:w 677, 10893, 10902, 10922
 - __iow_wrap_break_first:w 10893, 10899, 10905
 - __iow_wrap_break_loop:w 10893, 10908, 10916, 10920
 - __iow_wrap_break_none:w 10893, 10907, 10910
 - __iow_wrap_chunk:nw 10809, 10812, 10814, 10948, 10949, 10957, 10966, 10973
 - __iow_wrap_do: . 10773, 10778, 10778
 - __iow_wrap_end:n 10968, 10975
 - __iow_wrap_end_chunk:w 675, 10830, 10837, 10887, 10929
 - \c__iow_wrap_end_marker_tl 10699, 10783
 - __iow_wrap_fix_newline:w 10778, 10787, 10792, 10799
 - __iow_wrap_indent:n .. 10951, 10951
 - \c__iow_wrap_indent_marker_tl ... 10699, 10733
 - __iow_wrap_line:nw 675, 678, 10824, 10828, 10837, 10837, 10936
 - __iow_wrap_line_aux:Nw 10837, 10847, 10853
 - __iow_wrap_line_end:NnnnnnnN .. 10837, 10856, 10873
 - __iow_wrap_line_end:nw 677, 10837, 10878, 10881, 10913, 10914, 10923
 - __iow_wrap_line_loop:w 10837, 10841, 10844, 10850
 - __iow_wrap_line_seven:nnnnnnn .. 10837, 10868, 10872
 - \c__iow_wrap_marker_tl 671, 675, 10699, 10836
 - __iow_wrap_newline:n . 10968, 10968
 - \c__iow_wrap_newline_marker_tl .. 674, 10699, 10798
 - __iow_wrap_next:nw 10812, 10819, 10833, 10891, 10933
 - __iow_wrap_next_line:w 10885, 10926, 10926
 - __iow_wrap_start:w 10778, 10790, 10801
 - __iow_wrap_store_do:n 10884, 10971, 10978, 10981, 10981
 - \l_iow_wrap_tl 673, 674, 679, 10698, 10760, 10775, 10780, 10782, 10785, 10787, 10790, 10806, 10985, 10987
 - __iow_wrap_trim:N ... 679, 10914, 10945, 10971, 10978, 10993, 10995
 - __iow_wrap_trim:w 10993, 10996, 10997
 - __iow_wrap_trim_aux:w 10993, 10998, 10999
 - __iow_wrap_unindent:n . 10951, 10959
 - \c__iow_wrap_unindent_marker_tl . 10699, 10735
- J**
- \j 34892, 35775, 35945, 36024
 - \jcharwidowpenalty 1157
 - \jfam 1158
 - \jfont 1159
 - \jis 1160
 - \jobname 285
- K**
- \k 33333, 35799, 35823, 35898, 35899, 35916, 35917, 35939, 35940, 35941, 35996, 35997, 36022, 36023
 - \kanjiskip 1161
 - \kansuji 1162
 - \kansujichar 1163
 - \kcatcode 1164
 - \kchar 1203
 - \kchardef 1204

- \kern 286
- kernel internal commands:
 - __kernel_backend_align_begin: . 389
 - __kernel_backend_align_end: . 389
 - \g__kernel_backend_header_bool . 389
 - __kernel_backend_literal:n . 389
 - __kernel_backend_literal_pdf:n 389
 - __kernel_backend_literal_-
 - postscript:n 389
 - __kernel_backend_literal_svg:n 389
 - __kernel_backend_matrix:n . 389
 - __kernel_backend_postscript:n . 389
 - __kernel_backend_scope_begin: . 389
 - __kernel_backend_scope_end: . 389
 - __kernel_chk_cs_exist:N
 - 381, 1565,
 - 41787, 41788, 41789, 41805, 41845,
 - 42328, 42397, 42401, 42405, 42409
 - __kernel_chk_defined:NTF
 - 382, 2268, 2268, 2287,
 - 8445, 10371, 10615, 13518, 13553,
 - 19119, 19171, 24201, 31465, 31482
 - __kernel_chk_expr:nNnN
 - .. 382, 1567, 41886, 41888, 41897,
 - 41898, 42516, 42576, 42624, 42629,
 - 42659, 42665, 42683, 42697, 42701,
 - 42705, 42713, 42717, 42721, 42734
 - __kernel_chk_flag_exist:NN
 - 381, 41787,
 - 41790, 41814, 41846, 42316, 42342
 - __kernel_chk_if_free_cs:N
 - 633, 938,
 - 1973, 1973, 1981, 1982, 1988, 2052,
 - 8372, 12248, 12254, 13724, 17057,
 - 17385, 18302, 18323, 20142, 20144,
 - 20154, 20806, 20812, 21588, 22059,
 - 22151, 23789, 24013, 31306, 31312,
 - 31521, 31541, 31797, 36394, 42364
 - __kernel_chk_tl_type:NnnTF
 - ... 382, 878, 928, 980, 982, 7350,
 - 13551, 13551, 14438, 14445, 18173,
 - 19835, 21476, 21556, 21566, 26496
 - __kernel_chk_var_exist:N . 381,
 - 1565, 41787, 41787, 41796, 41832,
 - 41838, 41844, 42098, 42118, 42119
 - __kernel_chk_var_global:N
 - 381, 1565,
 - 41787, 41792, 41835, 41848, 42218
 - __kernel_chk_var_local:N
 - 381, 1565,
 - 41787, 41791, 41829, 41847, 42137
 - __kernel_chk_var_scope:NN
 - 381, 1565, 41787,
 - 41793, 41824, 41849, 42300, 42332,
 - 42336, 42341, 42347, 42351, 42355
 - __kernel_codepoint_case:nn
 - 388, 14391, 32572, 32572, 33831
 - __kernel_codepoint_data:nn
 - 388, 32093, 32464,
 - 32464, 32518, 32608, 32630, 32639
 - __kernel_codepoint_data:wn 12175,
 - 32084, 32512, 32600, 32617, 32621
 - __kernel_codepoint_to_bytes:n ..
 - 382, 15761,
 - 31942, 31974, 32002, 32002, 41600
 - __kernel_codepoint_to_grapheme_-
 - class:n 382, 32613,
 - 32615, 32625, 35180, 35225, 35283
 - __kernel_codepoint_to_wordbreak_-
 - class:n 382, 32613,
 - 32619, 32634, 35340, 35384, 35450
 - \l__kernel_color_stack_int 390
 - __kernel_cs_parm_from_arg_-
 - count:nnTF
 - 383, 1639, 2069, 2069, 2116
 - __kernel_debug_log:n
 - .. 382, 1567, 41878, 41880, 41884,
 - 41885, 42361, 42370, 42383, 42391
 - __kernel_dependency_version_-
 - check:Nn 383, 11697, 11697
 - __kernel_dependency_version_-
 - check:nn . 383, 11697, 11698, 11699
 - __kernel_deprecation_code:nn ...
 - 383, 1552, 1568,
 - 1581, 1583, 41377, 41403, 41410, 41411
 - __kernel_deprecation_error:Nnn .
 - 1552, 41380, 41413, 41413
 - __kernel_exp_not:w
 - 383, 433, 464, 482,
 - 738, 759, 979, 2725, 2725, 2727,
 - 2729, 2731, 2734, 2739, 4159, 12255,
 - 12281, 12282, 12289, 12290, 12304,
 - 12306, 12308, 12310, 12322, 12327,
 - 12332, 12338, 12339, 12346, 12347,
 - 12353, 12358, 12363, 12369, 12370,
 - 12377, 12378, 12394, 12398, 12403,
 - 12409, 12410, 12417, 12418, 12422,
 - 12426, 12431, 12437, 12438, 12445,
 - 12446, 12707, 13058, 13060, 13062,
 - 13064, 13150, 13155, 13164, 13359,
 - 13524, 13602, 13608, 13623, 13641,
 - 13679, 13730, 13735, 13740, 13745,
 - 18449, 21454, 21469, 22230, 31928,
 - 31967, 31981, 31998, 32989, 35508
 - \l__kernel_expl_bool
 - 105, 108, 122, 135, 1385

- \c__kernel_expl_date_tl
699, 1385, 11701, 11704, 11740, 11744
- __kernel_file_input_pop:
..... 383, 11511, 11551
- __kernel_file_input_push:n
..... 383, 11511, 11545
- __kernel_file_missing:n
.... 383, 10286, 11506, 11506, 11515
- __kernel_file_name_quote:n
..... 660, 10329, 10578,
11121, 11121, 11162, 11527, 40639
- __kernel_file_name_sanitizet:n ..
..... 383, 694, 10557, 11047,
11047, 11176, 11509, 11567, 11581
- __kernel_group_show:NN
..... 2309, 2310, 2312, 2313
- __kernel_if_debug:TF
.... 1562, 1562, 41391, 42779, 42779
- __kernel_int_add:nnn
..... 384, 18293, 18293, 30496
- __kernel_int_sep: 384, 3180,
4358, 10811, 13412, 13412, 13413,
14052, 18011, 18214, 19089, 19989,
21643, 22111, 23773, 24216, 32684
- __kernel_intarray_gset:Nnn
..... 384, 1052, 1054,
1057, 6687, 6793, 6796, 7494, 7566,
7568, 7574, 7582, 7584, 7587, 7708,
7710, 7714, 7716, 7728, 7731, 23877,
23878, 23948, 24018, 24030, 24055,
24055, 24070, 24138, 24186, 31218,
31219, 31221, 31225, 31226, 31227,
31544, 31545, 31579, 31582, 32401
- __kernel_intarray_gset_range_-
from_clist:Nnn 384,
6856, 23975, 23976, 24176, 24176
- __kernel_intarray_item:Nn . 384,
1053, 1058, 1242, 4406, 6804, 6831,
6915, 6916, 6940, 6941, 6948, 6955,
7012, 7016, 7035, 7571, 7762, 7981,
23901, 23926, 24107, 24107, 24119,
24153, 24172, 28748, 28754, 28757,
28760, 29497, 29500, 29503, 29506,
29509, 29512, 29515, 29518, 29521,
31267, 31268, 31269, 31637, 31640
- __kernel_intarray_range_to_-
clist:Nnn 384,
6785, 23960, 23961, 24157, 24157
- __kernel_ior_open:Nn 385,
660, 10293, 10310, 10310, 10321, 10344
- __kernel_iow_open:Nn
385, 10554, 10558, 10561, 10572, 10594
- __kernel_iow_with:Nnn . 385, 636,
669, 751, 9455, 9457, 9661, 9663,
10644, 10644, 10659, 13538, 13540
- \l__kernel_keyval_allow_blank_-
keys_bool 965,
20999, 21005, 21007, 22224, 22385
- __kernel_msg_error:nnn .. 9832, 9838
- __kernel_msg_error:nnnn . 9832, 9840
- __kernel_msg_error:nnnnn 9832, 9842
- __kernel_msg_expandable_-
error:nnn 9844, 9844, 9846
- __kernel_msg_expandable_-
error:nnnn 9844, 9848
- __kernel_msg_info:nnnn .. 9832, 9832
- __kernel_msg_log_eval:Nn
..... 385, 8438, 9823, 9825,
19067, 22031, 22142, 22210, 26571
- __kernel_msg_new:nnn
..... 9828, 9830, 10057
- __kernel_msg_new:nnnn ... 9828, 9828
- __kernel_msg_show_eval:Nn
..... 385, 8436, 9823, 9823,
19063, 22027, 22138, 22206, 26569
- __kernel_msg_warning:nnn 9832, 9834
- __kernel_msg_warning:nnnn 9832, 9836
- __kernel_patch:Nn
.... 42490, 42512, 42573, 42621,
42656, 42680, 42694, 42710, 42725
- __kernel_patch:nnn
1570, 42001, 42002, 42097, 42116,
42136, 42217, 42299, 42315, 42327,
42331, 42335, 42339, 42346, 42350,
42354, 42358, 42380, 42388, 42413,
42423, 42430, 42437, 42450, 42454,
42458, 42465, 42472, 42476, 42483
- __kernel_patch_aux:Nn . 42494, 42496
- __kernel_patch_aux:nnn
..... 42001, 42006, 42008
- __kernel_patch_cond:nn .. 42652,
42673, 42675, 42676, 42691, 42692
- __kernel_patch_deprecation:nnNNpn
..... 1552,
41373, 41373, 41431, 41434, 41454,
41457, 41460, 41464, 41466, 41470,
41476, 41486, 41488, 41490, 41492,
41495, 41497, 41499, 41501, 41503,
41505, 41512, 41514, 41516, 41518,
41520, 41522, 41524, 41526, 41528,
41532, 41534, 41536, 41538, 41540,
41543, 41545, 41547, 41549, 41551,
41553, 41556, 41559, 41563, 41566,
41569, 41572, 41575, 41578, 41581,
41583, 41585, 41587, 41592, 41594,
41596, 41599, 41601, 41603, 41605,
41607, 41609, 41611, 41613, 41615,
41617, 41619, 41621, 41623, 41625,

41627, 41629, 41631, 41635, 41637,
 41648, 41654, 41660, 41668, 41670
 __kernel_patch_eval:nn
 42508, 42524, 42536, 42547,
 42558, 42569, 42584, 42588, 42598,
 42605, 42612, 42617, 42637, 42644
 __kernel_patch_weird:nnn
 1571, 42001, 42064,
 42366, 42396, 42400, 42404, 42408
 __kernel_patch_weird_aux:nnn ...
 42001, 42068, 42070
 __kernel_pdf_object_id:n
 385, 40893, 40910
 __kernel_pdf_object_id_indexed:nn
 385, 40972, 40990
 \g__kernel_prg_map_int . 385, 477,
 735, 894, 992, 1385, 3975, 3977,
 3986, 4106, 4108, 4116, 4128, 8713,
 10469, 10471, 10478, 12955, 12957,
 12959, 12964, 13951, 13953, 13957,
 13962, 17960, 17961, 17967, 17968,
 18019, 18021, 18023, 18025, 18675,
 18678, 18686, 18689, 18700, 19585,
 19587, 19589, 19594, 21403, 21404,
 21409, 21411, 21866, 21869, 21873,
 21876, 21887, 26858, 26861, 26865,
 26868, 26879, 35484, 35486, 35488,
 35490, 35494, 35496, 35498, 35500
 __kernel_primitive:NN
 355, 143, 143, 147, 148, 149, 150,
 151, 152, 153, 154, 155, 156, 157,
 158, 159, 160, 161, 162, 163, 164,
 165, 166, 167, 168, 169, 170, 171,
 172, 173, 174, 175, 176, 177, 178,
 179, 180, 181, 182, 183, 184, 185,
 186, 187, 188, 189, 190, 191, 192,
 193, 194, 195, 196, 197, 198, 199,
 200, 201, 202, 203, 204, 205, 206,
 207, 208, 209, 210, 211, 212, 213,
 214, 215, 216, 217, 218, 219, 220,
 221, 222, 223, 224, 225, 226, 227,
 228, 229, 230, 231, 232, 233, 234,
 235, 236, 237, 238, 239, 240, 241,
 242, 243, 244, 245, 246, 247, 248,
 249, 250, 251, 252, 253, 254, 255,
 256, 257, 258, 259, 260, 261, 262,
 263, 264, 265, 266, 267, 268, 269,
 270, 271, 272, 273, 274, 275, 276,
 277, 278, 279, 280, 281, 282, 283,
 284, 285, 286, 287, 288, 289, 290,
 291, 292, 293, 294, 295, 296, 297,
 298, 299, 300, 301, 302, 303, 304,
 305, 306, 307, 308, 309, 310, 311,
 312, 313, 314, 315, 316, 317, 318,
 319, 320, 321, 322, 323, 324, 325,
 326, 327, 328, 329, 330, 331, 332,
 333, 334, 335, 336, 337, 338, 339,
 340, 341, 342, 343, 344, 345, 346,
 347, 348, 349, 350, 351, 352, 353,
 354, 355, 356, 357, 358, 359, 360,
 361, 362, 363, 364, 365, 366, 367,
 368, 369, 370, 371, 372, 373, 374,
 375, 376, 377, 378, 379, 380, 381,
 382, 383, 384, 385, 386, 387, 388,
 389, 390, 391, 392, 393, 394, 395,
 396, 397, 398, 399, 400, 401, 402,
 403, 404, 405, 406, 407, 408, 409,
 410, 411, 412, 413, 414, 415, 416,
 417, 418, 419, 420, 421, 422, 423,
 424, 425, 426, 427, 428, 429, 430,
 431, 432, 433, 434, 435, 436, 437,
 438, 439, 440, 441, 442, 443, 444,
 445, 446, 447, 448, 449, 450, 451,
 452, 453, 454, 455, 456, 457, 458,
 459, 460, 461, 462, 463, 464, 465,
 466, 467, 468, 469, 470, 471, 472,
 473, 474, 475, 476, 477, 478, 479,
 480, 481, 482, 483, 484, 485, 486,
 487, 488, 489, 490, 491, 492, 493,
 494, 495, 496, 497, 498, 499, 500,
 501, 502, 503, 504, 505, 506, 507,
 508, 509, 510, 511, 512, 513, 514,
 515, 516, 517, 518, 519, 520, 521,
 522, 523, 524, 525, 526, 527, 528,
 529, 530, 531, 532, 533, 534, 535,
 536, 537, 538, 539, 540, 541, 542,
 543, 544, 545, 546, 547, 548, 549,
 550, 551, 552, 553, 554, 555, 556,
 557, 558, 559, 560, 561, 562, 564,
 565, 566, 567, 568, 569, 570, 571,
 572, 573, 575, 576, 577, 578, 579,
 580, 581, 582, 583, 584, 585, 586,
 587, 588, 589, 590, 591, 592, 593,
 594, 595, 596, 597, 598, 599, 600,
 601, 602, 603, 604, 605, 606, 607,
 609, 611, 613, 614, 615, 616, 617,
 618, 619, 620, 621, 622, 623, 624,
 625, 626, 628, 629, 630, 631, 632,
 633, 634, 635, 636, 637, 638, 639,
 640, 641, 642, 643, 644, 645, 646,
 647, 648, 649, 650, 651, 652, 653,
 654, 655, 656, 657, 658, 659, 660,
 661, 662, 663, 664, 665, 666, 667,
 668, 669, 670, 671, 672, 673, 674,
 675, 676, 677, 678, 679, 680, 681,
 682, 683, 684, 689, 698, 699, 700,
 701, 702, 703, 705, 706, 707, 708,
 709, 710, 711, 712, 713, 714, 715,

717, 719, 721, 722, 723, 725, 726,
727, 728, 729, 730, 732, 734, 735,
737, 739, 740, 741, 742, 743, 744,
745, 746, 747, 748, 749, 750, 751,
752, 753, 754, 755, 756, 757, 758,
759, 760, 761, 762, 763, 764, 766,
768, 769, 770, 771, 772, 773, 774,
775, 776, 777, 778, 779, 780, 781,
782, 783, 785, 786, 788, 789, 790,
791, 792, 793, 794, 795, 796, 797,
799, 800, 802, 803, 804, 805, 806,
808, 809, 810, 811, 812, 813, 814,
815, 816, 817, 818, 819, 820, 821,
822, 823, 825, 826, 827, 828, 829,
830, 831, 832, 833, 834, 835, 836,
837, 838, 839, 840, 841, 842, 843,
844, 845, 846, 847, 848, 849, 850,
851, 852, 853, 854, 855, 856, 857,
858, 859, 860, 861, 862, 863, 864,
865, 866, 867, 868, 869, 870, 871,
872, 873, 874, 875, 876, 878, 880,
881, 882, 883, 884, 885, 886, 887,
888, 889, 890, 891, 892, 893, 894,
895, 896, 897, 898, 899, 900, 901,
902, 903, 904, 905, 906, 907, 908,
909, 910, 911, 912, 913, 914, 915,
916, 917, 918, 919, 920, 922, 923,
924, 925, 926, 927, 928, 929, 930,
931, 932, 933, 934, 935, 936, 937,
938, 939, 940, 942, 944, 946, 947,
948, 949, 950, 951, 952, 953, 954,
955, 956, 957, 958, 959, 960, 961,
962, 963, 964, 965, 966, 967, 968,
969, 970, 971, 972, 973, 974, 975,
976, 977, 978, 979, 980, 981, 982,
983, 984, 985, 986, 987, 988, 989,
990, 992, 994, 995, 996, 997, 999,
1000, 1001, 1002, 1004, 1005, 1007,
1009, 1010, 1011, 1012, 1013, 1015,
1017, 1018, 1019, 1020, 1022, 1023,
1024, 1025, 1026, 1027, 1028, 1029,
1030, 1031, 1032, 1033, 1034, 1035,
1036, 1037, 1038, 1039, 1040, 1041,
1042, 1043, 1044, 1045, 1046, 1047,
1048, 1049, 1050, 1051, 1052, 1053,
1054, 1055, 1056, 1057, 1058, 1059,
1061, 1063, 1064, 1066, 1068, 1069,
1070, 1071, 1073, 1074, 1075, 1077,
1079, 1081, 1082, 1083, 1084, 1085,
1086, 1087, 1088, 1089, 1090, 1091,
1092, 1094, 1096, 1097, 1098, 1099,
1100, 1101, 1102, 1103, 1104, 1105,
1106, 1107, 1108, 1109, 1110, 1111,
1112, 1114, 1116, 1117, 1118, 1119,
1120, 1121, 1122, 1123, 1124, 1125,
1126, 1127, 1128, 1129, 1130, 1131,
1132, 1133, 1134, 1135, 1136, 1137,
1138, 1139, 1140, 1141, 1142, 1143,
1144, 1145, 1146, 1147, 1148, 1149,
1150, 1151, 1152, 1153, 1154, 1155,
1156, 1157, 1158, 1159, 1160, 1161,
1162, 1163, 1164, 1165, 1166, 1167,
1168, 1169, 1170, 1171, 1172, 1173,
1174, 1175, 1176, 1177, 1178, 1179,
1180, 1181, 1183, 1185, 1186, 1187,
1188, 1189, 1191, 1192, 1193, 1194,
1195, 1196, 1197, 1198, 1199, 1200,
1201, 1202, 1203, 1204, 1205, 1206,
1207, 1208, 1209, 1210, 1211, 1212,
1213, 1214, 1215, 1216, 1217, 1218
__kernel_quark_new_conditional:Nn
..... 387, 4449, 11042, 12467,
17157, 17177, 19867, 22492, 32665
__kernel_quark_new_test:N
..... 386, 849, 850, 852, 853,
8414, 11045, 11046, 12466, 13689,
13690, 17157, 17157, 18209, 18210,
20745, 32670, 32671, 35505, 41686
__kernel_randint:n
..... 387, 1270, 1274,
30303, 30303, 30315, 30473, 30561
__kernel_randint:nn
.... 387, 30477, 30481, 30481, 30559
\\c__kernel_randint_max_int
.... 1274, 1385, 30302, 30471, 30558
__kernel_register_log:N
..... 387, 2277,
2281, 2283, 2284, 19064, 22028,
22029, 22139, 22140, 22207, 22208
__kernel_register_show:N
..... 387, 751, 2277, 2277,
2279, 2280, 19060, 22024, 22135, 22203
__kernel_register_show_aux:NN ..
..... 2277, 2278, 2282, 2285
__kernel_register_show_aux:nNN ..
..... 2277, 2289, 2293
__kernel_show:NN
..... 2295, 2295, 2298, 2301
__kernel_str_to_other:n
.. 388, 765, 767, 772, 838, 14007,
14007, 14060, 14121, 16655, 16807
__kernel_str_to_other_fast:n ...
... 388, 4658, 5869, 10686, 10782,
13958, 13978, 14030, 14030, 14652
__kernel_str_to_other_fast_-
loop:w 14030
__kernel_sys_configuration_-
load:n 622, 8876, 8942

- _kernel_sys_everyjob: 387, [9061](#), 9061, 9238
- _kernel_tl_gset:Nn .. 388, [584](#),
585, [714](#), 759, 3294, 3852, 4657,
5867, 7603, 7614, 7678, 7822, 9262,
[12244](#), [12245](#), [12287](#), [12308](#), [12310](#),
[12352](#), [12357](#), [12362](#), [12367](#), [12375](#),
[12422](#), [12425](#), [12430](#), [12435](#), [12443](#),
[12654](#), [12658](#), [13084](#), [13086](#), [13088](#),
[13379](#), [13655](#), [13721](#), [13734](#), [13744](#),
[13757](#), [13761](#), [14586](#), [14602](#), [14652](#),
[14663](#), [14782](#), [14834](#), [14845](#), [15003](#),
[15052](#), [15105](#), [15111](#), [15345](#), [15545](#),
[15702](#), [17421](#), [17426](#), [17444](#), [17448](#),
[17489](#), [17555](#), [17595](#), [17624](#), [17630](#),
[17689](#), [17838](#), [17881](#), [18071](#), [18081](#),
[19258](#), [19285](#), [19304](#), [19353](#), [19389](#),
[19432](#), [19471](#), [26460](#), [42135](#), [42275](#)
- _kernel_tl_set:Nn 388, [4547](#), [5437](#),
[5442](#), [5713](#), [5782](#), [7757](#), [7790](#), [10317](#),
[10556](#), [10568](#), [10685](#), [10760](#), [10763](#),
[10764](#), [10780](#), [10785](#), [10944](#), [10964](#),
[10983](#), [10985](#), [11291](#), [11418](#), [11433](#),
[12244](#), [12244](#), [12279](#), [12304](#), [12306](#),
[12321](#), [12326](#), [12331](#), [12336](#), [12344](#),
[12394](#), [12397](#), [12402](#), [12407](#), [12415](#),
[12652](#), [12656](#), [13078](#), [13080](#), [13082](#),
[13377](#), [13650](#), [13667](#), [13719](#), [13729](#),
[13739](#), [13755](#), [13759](#), [14541](#), [17411](#),
[17416](#), [17442](#), [17446](#), [17468](#), [17487](#),
[17547](#), [17593](#), [17622](#), [17628](#), [17687](#),
[17794](#), [17819](#), [17836](#), [17850](#), [17878](#),
[18069](#), [18079](#), [19256](#), [19283](#), [19302](#),
[19351](#), [19387](#), [19430](#), [19469](#), [23171](#),
[23270](#), [23519](#), [26458](#), [41733](#), [41922](#),
[41924](#), [41926](#), [42134](#), [42193](#), [42417](#)
- _kernel_tl_to_str:w
..... 388, [733](#), [759](#),
[1412](#), [1414](#), [12798](#), [12892](#), [13005](#),
[13719](#), [13721](#), [13725](#), [13730](#), [13735](#),
[13740](#), [13745](#), [13933](#), [14001](#), [17155](#)
- keys commands:
- \l_keys_choice_int
247, 250, 252, 253, [22457](#), [22526](#),
[22689](#), [22692](#), [22698](#), [22699](#), [23159](#)
- \l_keys_choice_str 247, 250,
252, 253, [22457](#), [22520](#), [22696](#), [23153](#)
- \l_keys_choice_tl
.. [22459](#), [22519](#), [22697](#), [23152](#), [40446](#)
- \keys_define:nn
246, [10022](#), [22508](#), [22508](#), [22529](#), [40434](#)
- \keys_if_choice_exist:nnn [23635](#)
- \keys_if_choice_exist:nnnTF
..... 258, [23635](#)
- \keys_if_choice_exist_p:nnn
..... 258, [23635](#)
- \keys_if_exist:nn [23620](#), [23626](#)
- \keys_if_exist:nnTF
..... 258, [1045](#), [23620](#), [23672](#)
- \keys_if_exist_p:nn 258, [23620](#)
- \l_keys_key_str 254, 257,
[22464](#), [22522](#), [22757](#), [22758](#), [23155](#),
[23279](#), [23284](#), [23286](#), [23399](#), [23403](#),
[23438](#), [23441](#), [23442](#), [23499](#), [23556](#)
- \l_keys_key_tl [22465](#),
[22521](#), [22757](#), [22758](#), [23154](#), [23286](#)
- \keys_log:nn 258, [23656](#), [23658](#)
- \l_keys_path_str 254, [1019](#),
[22469](#), [22524](#), [22549](#), [22567](#), [22586](#),
[22603](#), [22639](#), [22641](#), [22643](#), [22646](#),
[22658](#), [22661](#), [22665](#), [22673](#), [22675](#),
[22676](#), [22679](#), [22694](#), [22711](#), [22724](#),
[22728](#), [22739](#), [22742](#), [22750](#), [22756](#),
[22760](#), [22763](#), [22767](#), [22778](#), [22780](#),
[22782](#), [22785](#), [22799](#), [22804](#), [22806](#),
[22821](#), [22831](#), [22837](#), [22841](#), [22856](#),
[22865](#), [22909](#), [22910](#), [22928](#), [22971](#),
[23157](#), [23270](#), [23278](#), [23280](#), [23332](#),
[23335](#), [23371](#), [23375](#), [23380](#), [23396](#),
[23414](#), [23416](#), [23417](#), [23421](#), [23430](#),
[23479](#), [23509](#), [23532](#), [23544](#), [23553](#)
- \l_keys_path_tl
.. [22470](#), [22523](#), [22586](#), [22665](#), [23156](#)
- \keys_precompile:nnN 257, [23250](#), [23250](#)
- \keys_set:nn .. 246, 248, 249, 254-
257, [23182](#), [23182](#), [23192](#), [23254](#), [40599](#)
- \keys_set_exclude_groups:nnn
257, [23210](#), [23224](#), [23229](#), [41454](#), [41455](#)
- \keys_set_exclude_groups:nnnN ...
257, [23210](#), [23221](#), [23223](#), [41457](#), [41458](#)
- \keys_set_exclude_groups:nnnnN ..
..... 257, [23210](#), [23210](#),
[23220](#), [23222](#), [23226](#), [41460](#), [41461](#)
- \keys_set_filter:nnn
..... [41454](#), [41455](#), [41456](#)
- \keys_set_filter:nnnN
..... [41454](#), [41458](#), [41459](#)
- \keys_set_filter:nnnnN
..... [41454](#), [41461](#), [41462](#)
- \keys_set_groups:nnn
..... 257, [23210](#), [23244](#), [23249](#)
- \keys_set_groups:nnnN
..... 257, [23210](#), [23241](#), [23243](#)
- \keys_set_groups:nnnnN
257, [23210](#), [23230](#), [23240](#), [23242](#), [23246](#)
- \keys_set_known:nn
..... 256, [23193](#), [23207](#), [23209](#)

- \keys_set_known:nnN 256, [23193](#), [23204](#), [23206](#)
- \keys_set_known:nnnN [256](#), [23193](#), [23193](#), [23203](#), [23205](#), [23208](#)
- \keys_show:nn [258](#), [23656](#), [23656](#)
- \l_keys_usage_load_prop [254](#), [22486](#), [22875](#), [22882](#), [22889](#)
- \l_keys_usage_preamble_prop [254](#), [22486](#), [22877](#), [22884](#), [22891](#)
- \l_keys_value_tl [254](#), [22480](#), [22525](#), [22759](#), [22856](#), [23158](#), [23374](#), [23378](#), [23390](#), [23402](#), [23417](#), [23442](#), [23471](#), [23475](#), [23501](#), [23511](#), [23539](#)
- keys internal commands:
 - __keys_bool_set:Nn [22629](#), [22629](#), [22631](#), [22650](#), [22943](#), [22945](#), [22947](#), [22949](#)
 - __keys_bool_set:Nnn [22629](#), [22630](#), [22633](#), [22635](#)
 - __keys_bool_set_inverse:Nn [22629](#), [22632](#), [22634](#), [22951](#), [22953](#), [22955](#), [22957](#)
 - __keys_check_forbidden: [22824](#), [22851](#)
 - __keys_check_groups: . [23336](#), [23344](#)
 - __keys_check_required: [22824](#), [22860](#)
 - \c__keys_check_root_str .. [22450](#), [22831](#), [22837](#), [22841](#), [23416](#), [23441](#)
 - __keys_choice_find:n [22652](#), [23550](#), [23550](#), [23566](#)
 - __keys_choice_find:nn [23550](#), [23553](#), [23555](#), [23559](#)
 - __keys_choice_make: [22638](#), [22651](#), [22651](#), [22683](#), [22777](#), [22959](#)
 - __keys_choice_make:N [22651](#), [22652](#), [22654](#), [22655](#)
 - __keys_choice_make_aux:N [22651](#), [22667](#), [22669](#), [22671](#)
 - __keys_choices_make:nn [22682](#), [22682](#), [22961](#), [22963](#), [22965](#), [22967](#), [22969](#)
 - __keys_choices_make:Nnn [22682](#), [22683](#), [22685](#), [22686](#)
 - __keys_cmd_set:nn . [22639](#), [22641](#), [22693](#), [22704](#), [22704](#), [22706](#), [22778](#), [22780](#), [22782](#), [22806](#), [22928](#), [22971](#)
 - __keys_cmd_set_direct:nn [22643](#), [22675](#), [22676](#), [22704](#), [22705](#), [22707](#), [22798](#), [42386](#)
 - \c__keys_code_root_str [1042](#), [22450](#), [22708](#), [22711](#), [22760](#), [23414](#), [23438](#), [23463](#), [23489](#), [23561](#), [23630](#), [23647](#), [23680](#), [42382](#)
 - __keys_cs_set:NNpn [22709](#), [22709](#), [22718](#), [22981](#), [22983](#), [22985](#), [22987](#), [22989](#), [22991](#), [22993](#), [22995](#)
 - __keys_cs_undefine:N ... [22500](#), [22500](#), [22723](#), [22738](#), [22820](#), [22840](#)
 - __keys_default_inherit: [23367](#), [23381](#), [23393](#)
 - \c__keys_default_root_str [22450](#), [22724](#), [22728](#), [23371](#), [23375](#), [23399](#), [23403](#), [23468](#), [23472](#)
 - __keys_default_set:n [22648](#), [22719](#), [22719](#), [22787](#), [22997](#), [22999](#), [23001](#), [23003](#), [23005](#)
 - __keys_define:n . [22515](#), [22530](#), [22530](#)
 - __keys_define:nn [22515](#), [22530](#), [22535](#)
 - __keys_define_aux:nn [22530](#), [22533](#), [22538](#), [22540](#)
 - __keys_define_code:n [22544](#), [22594](#), [22594](#)
 - __keys_define_code:nn [22594](#), [22598](#), [22608](#)
 - __keys_define_code:w [22594](#), [22610](#), [22617](#)
 - \l_keys_exclude_bool [22475](#), [23146](#), [23188](#), [23199](#), [23216](#), [23236](#), [23339](#), [23357](#), [23362](#)
 - __keys_execute: [23290](#), [23340](#), [23359](#), [23363](#), [23412](#), [23412](#)
 - __keys_execute:nn [22763](#), [23412](#), [23417](#), [23442](#), [23475](#), [23483](#), [23484](#), [23485](#), [23562](#), [23563](#)
 - __keys_execute_inherit: [22768](#), [23412](#), [23422](#), [23426](#)
 - __keys_execute_inherit:n [23412](#), [23431](#), [23435](#), [23451](#)
 - __keys_execute_unknown: [23412](#), [23423](#), [23433](#), [23457](#)
 - \l_keys_exp_str [22460](#), [23328](#), [23385](#), [23387](#)
 - __keys_find_key_module:wNN ... [22755](#), [22804](#), [23258](#), [23278](#), [23294](#)
 - __keys_find_key_module_auxi:Nw . [23258](#), [23296](#), [23299](#), [23307](#), [23312](#)
 - __keys_find_key_module_auxii:Nw [23258](#), [23296](#), [23303](#), [23304](#)
 - __keys_find_key_module_auxiii:Nn [23258](#)
 - __keys_find_key_module_auxiiii:Nw [23307](#), [23309](#)
 - __keys_find_key_module_auxiv:Nw [23258](#), [23297](#), [23314](#), [23316](#)
 - __keys_find_key_module_auxv:Nw . [23258](#), [23318](#), [23323](#)
 - \l_keys_groups_clist [1039](#), [22461](#), [22735](#), [22736](#), [22743](#), [23334](#), [23349](#)


```

\c__keys_groups_root_str .....
.. 22450, 22739, 22742, 23332, 23335
\__keys_groups_set:n .....
..... 22733, 22733, 23023
\__keys_if_exist:nn .....
..... 23620, 23622, 23627, 23634
\__keys_if_exist:nnn .....
..... 23635, 23638, 23643, 23655
\__keys_inherit:n 22746, 22746, 23025
\l__keys_inherit_bool .....
.. 22462, 23428, 23432, 23444, 23452
\l__keys_inherit_clist .....
..... 22463, 22749, 22751
\c__keys_inherit_root_str .....
.... 22450, 22750, 22767, 23380,
23396, 23421, 23430, 23448, 23450
\l__keys_inherit_str .....
..... 22471, 22518, 22762,
23151, 23277, 23440, 23552, 23556
\__keys_initialize:n 22753, 22753,
23027, 23029, 23031, 23033, 23035
\__keys_legacy_if_inverse:nn . 22771
\__keys_legacy_if_inverse:nnnn 22771
\__keys_legacy_if_set:nn .....
..... 22771, 22771, 23045, 23047
\__keys_legacy_if_set:nnnn .....
..... 22772, 22774, 22775
\__keys_legacy_if_set_inverse:nn
..... 22773, 23049, 23051
\__keys_meta_make:n .....
..... 22794, 22794, 23053
\__keys_meta_make:nn .....
..... 22794, 22795, 22796, 23055
\l__keys_module_str 22466, 22514,
22517, 22569, 22795, 22904, 22915,
23175, 23178, 23180, 23261, 23266,
23276, 23279, 23282, 23283, 23291,
23463, 23468, 23472, 23475, 23479
\__keys_multichoice_find:n .....
..... 22654, 23550, 23565
\__keys_multichoice_make: .....
..... 22651, 22653, 22685, 23057
\__keys_multichoices_make:nn ...
..... 22682, 22684,
23059, 23061, 23063, 23065, 23067
\l__keys_no_value_bool .....
..... 22467, 22532, 22537,
22596, 22853, 22862, 23260, 23265,
23369, 23465, 23500, 23510, 23538
\l__keys_only_known_bool .....
..... 22468, 23145,
23187, 23198, 23215, 23235, 23459
\__keys_parent:n .....
22658, 22661, 22665, 22767, 23380,
23396, 23421, 23430, 23567, 23567
\__keys_parent_auxi:w .....
.. 23567, 23569, 23572, 23578, 23582
\__keys_parent_auxii:w .....
..... 23567, 23569, 23576
\__keys_parent_auxiii:n .....
..... 23567, 23578, 23580
\__keys_parent_auxiv:w .....
..... 23567, 23570, 23584
\__keys_precompile:n .....
..... 22493, 22493, 22705,
22713, 23694, 23696, 23702, 23703
\l__keys_precompile_bool .....
..... 22484, 22495, 23252, 23255
\l__keys_precompile_tl .....
..... 22484, 22496, 23253, 23256
\__keys_prop_put:Nn 22801, 22801,
22814, 23077, 23079, 23081, 23083
\__keys_property_find:n .....
..... 22542, 22553, 22553
\__keys_property_find_auxi:w ...
..... 22553, 22555, 22559, 22573
\__keys_property_find_auxii:w ...
..... 22553, 22556, 22563, 22564
\__keys_property_find_auxiii:w ..
..... 22553, 22573, 22576, 22581
\__keys_property_find_auxiv:w ...
... 1019, 22553, 22574, 22580, 22582
\__keys_property_find_err:w ....
.. 22553, 22557, 22565, 22588, 22589
\l__keys_property_str ... 22474,
22543, 22546, 22549, 22585, 22591,
22599, 22600, 22603, 22606, 22611
\c__keys_props_root_str .....
..... 22456, 22543, 22599,
22600, 22606, 22942, 22944, 22946,
22948, 22950, 22952, 22954, 22956,
22958, 22960, 22962, 22964, 22966,
22968, 22970, 22972, 22974, 22976,
22978, 22980, 22982, 22984, 22986,
22988, 22990, 22992, 22994, 22996,
22998, 23000, 23002, 23004, 23006,
23008, 23010, 23012, 23014, 23016,
23018, 23020, 23022, 23024, 23026,
23028, 23030, 23032, 23034, 23036,
23038, 23040, 23042, 23044, 23046,
23048, 23050, 23052, 23054, 23056,
23058, 23060, 23062, 23064, 23066,
23068, 23070, 23072, 23074, 23076,
23078, 23080, 23082, 23084, 23086,
23088, 23090, 23092, 23094, 23096,
23098, 23100, 23102, 23104, 23106,
23108, 23110, 23112, 23114, 23116,
23118, 23120, 23122, 23124, 23126,

```


- 23128, 23130, 41438, 41440, 41442,
41444, 41446, 41448, 41450, 41452
 - __keys_quark_if_no_value:N .. 22492
 - __keys_quark_if_no_value:NTF ...
..... 22492, 23495
 - __keys_quark_if_no_value_p:N . 22492
 - \l_keys_relative_tl 22472,
23140, 23150, 23495, 23505, 23519,
23520, 23524, 23525, 23533, 23545
 - __keys_reset_bool:N
.. 23132, 23145, 23146, 23147, 23162
 - __keys_reset_var:N 22517, 22518,
22519, 22520, 22521, 22522, 22523,
22524, 22525, 23132, 23148, 23149,
23150, 23151, 23152, 23153, 23154,
23155, 23156, 23157, 23158, 23168
 - \l_keys_selective_bool
..... 22475, 23147,
23189, 23200, 23217, 23237, 23288
 - \l_keys_selective_clist
... 1039, 22477, 23139, 23149, 23347
 - __keys_set:nn
..... 22799, 23132, 23142, 23174
 - __keys_set:nnn . 23132, 23175, 23176
 - __keys_set:nnnnNn 23132,
23132, 23184, 23195, 23212, 23232
 - __keys_set_keyval:n
..... 23179, 23258, 23258
 - __keys_set_keyval:nn
..... 23179, 23258, 23263
 - __keys_set_keyval:nnn
.. 23258, 23261, 23266, 23268, 23293
 - __keys_set_selective:
..... 23258, 23289, 23330
 - __keys_show:n .. 23656, 23676, 23690
 - __keys_show:Nnn
..... 23656, 23657, 23659, 23660
 - __keys_show:Nw . 23656, 23709, 23713
 - __keys_show:w .. 23656, 23692, 23701
 - __keys_show_aux:Nnn
..... 23656, 23662, 23667, 23689
 - __keys_store_unused: ... 23341,
23358, 23364, 23412, 23460, 23493
 - __keys_store_unused:w
..... 23523, 23544, 23549
 - __keys_store_unused_aux:
..... 23412, 23514, 23517
 - __keys_tmp:w 23588, 23600
 - \l_keys_tmp_bool
..... 22481, 23346, 23351, 23355
 - \l_keys_tmp_clist
.. 22478, 23185, 23208, 23227, 23247
 - \l_keys_tmpa_tl ... 22481, 22805,
22904, 22905, 22913, 22914, 22916
 - \l_keys_tmpb_tl 22481,
22805, 22810, 22906, 22913, 22914
 - __keys_trim_spaces:n
..... 1019, 22514, 22570,
22694, 23178, 23274, 23520, 23561,
23562, 23587, 23590, 23623, 23624,
23639, 23640, 23641, 23664, 23665
 - __keys_trim_spaces_auxi:w
.. 23587, 23592, 23593, 23602, 23612
 - __keys_trim_spaces_auxii:w 23587,
23594, 23596, 23606, 23613, 23615
 - __keys_trim_spaces_auxiii:w ...
..... 23587, 23597, 23610, 23616
 - \c_keys_type_root_str
..... 22450, 22658, 22661, 22673
 - __keys_undefine:
..... 22748, 22815, 22815, 23125
 - \l_keys_unused_clist 22479, 23138,
23143, 23148, 23497, 23507, 23535
 - __keys_usage:n . 22869, 22869, 23127
 - __keys_usage:NN
..... 22869, 22875, 22877,
22882, 22884, 22889, 22891, 22902
 - __keys_usage:w . 22869, 22908, 22918
 - __keys_usage_aux:w
..... 22869, 22922, 22924
 - __keys_value_or_default:n
..... 23287, 23367, 23367
 - __keys_value_requirement:nn ...
..... 22730,
22824, 22824, 22939, 23129, 23131
 - __keys_value_set:NN
..... 23367, 23410, 23411
 - __keys_value_set:Nn
..... 23367, 23408, 23409
 - __keys_variable_set:NnnN
..... 22925, 22925,
22935, 22938, 22973, 22975, 22977,
22979, 23093, 23095, 23097, 23099,
23101, 23103, 23105, 23107, 23109,
23111, 23113, 23115, 23117, 23119,
23121, 23123, 41439, 41441, 41443,
41445, 41447, 41449, 41451, 41453
 - __keys_variable_set_required:NnnN
..... 22925, 22936, 22941,
23007, 23009, 23011, 23013, 23015,
23017, 23019, 23021, 23037, 23039,
23041, 23043, 23069, 23071, 23073,
23075, 23085, 23087, 23089, 23091
- keyval commands:
- \keyval_map_break:
... 261, 22445, 22447, 22447, 22448
 - \keyval_map_break:n 261, 22447, 22448

- \keyval_map_inline:nnn 260, 22437, 22437
- \keyval_parse:NNn 260, 1013, 1014, 22228, 22238, 22352, 22444, 22515, 23179
- \keyval_parse:nnn 259, 260, 965, 1007, 1012, 21006, 22228, 22228, 22238, 22353
- keyval internal commands:
 - __keyval_blank_key_error:w 22359, 22368, 22381, 22383
 - __keyval_blank_true:w 22313, 22381, 22381
 - __keyval_clean_up_active:w 1010, 22255, 22268, 22289, 22289, 22321, 22341
 - __keyval_clean_up_other:w 1010, 22294, 22299, 22310, 22310
 - __keyval_end_loop_active:w 22242, 22335, 22343
 - __keyval_end_loop_other:w 1011, 22252, 22335, 22335
 - __keyval_if_blank:w 22313, 22359, 22368, 22378, 22379
 - __keyval_if_empty:w 22378, 22378, 22379
 - __keyval_if_recursion_tail:w ... 22241, 22251, 22378, 22380
 - __keyval_key:nn 1010, 22315, 22354, 22366, 22381
 - __keyval_loop_active:nnw 22233, 22239, 22239, 22342
 - __keyval_loop_other:nnw 1008, 22243, 22249, 22249, 22325, 22333, 22345, 22364, 22373, 22382, 22383, 22388
 - __keyval_misplaced_equal_after_active_error:w 1009, 22262, 22266, 22317, 22317
 - __keyval_misplaced_equal_in_split_error:w 22273, 22278, 22282, 22286, 22302, 22307, 22317, 22327
 - __keyval_pair:nnnn 1009, 1010, 22288, 22309, 22354, 22357
 - __keyval_split_active:w 1008, 1009, 22245, 22247, 22253, 22272, 22337
 - __keyval_split_active_auxi:w ... 22254, 22259, 22259, 22290, 22340
 - __keyval_split_active_auxii:w .. 1009, 22259, 22263, 22265, 22319
 - __keyval_split_active_auxiii:w . 1009, 22259, 22269, 22270
 - __keyval_split_active_auxiv:w .. 1009, 22259, 22274, 22277
 - __keyval_split_active_auxv:w ... 22259, 22283, 22285
 - __keyval_split_other:w .. 22245, 22245, 22261, 22281, 22292, 22301
 - __keyval_split_other_auxi:w ... 1010, 22293, 22296, 22296, 22311
 - __keyval_split_other_auxii:w ... 22296, 22297, 22298
 - __keyval_split_other_auxiii:w .. 1010, 22296, 22303, 22306
 - __keyval_tmp:n 22397, 22435, 22440, 22444
 - __keyval_tmp:nn 1011, 22226, 22350, 22355, 22376, 22441, 22444
 - __keyval_trim:nN 22269, 22288, 22297, 22309, 22315, 22381, 22396, 22399
 - __keyval_trim_auxi:w 22396, 22401, 22410, 22413, 22418
 - __keyval_trim_auxii:w 22396, 22405, 22418
 - __keyval_trim_auxiii:w .. 22396, 22406, 22420, 22423, 22427, 22431
 - __keyval_trim_auxiv:w 22396, 22408, 22429
 - \knacode 672
 - \knbccode 673
 - \knbscode 674
 - \kuten 1165
- L**
- \L 33342, 34883, 35765
- \l 33342, 34883, 35777
- \label 32936, 32946, 35738
- \language 287
- \lastallocatedtoks 3255
- \lastbox 288
- \lastkern 289
- \lastlinefit 505
- \lastnamedcs 841
- \lastnodechar 1166
- \lastnodefont 1167
- \lastnodesubtype 1168
- \lastnodetype 506
- \lastpenalty 290
- \lastsavedboxresourceindex 940
- \lastsavedimageresourceindex 942
- \lastsavedimageresourcepages 944
- \lastskip 291
- \lastxpos 946
- \lastypos 947
- \latelua 842

- \lateluafunction 843
 - \lccode 64, 65, 292
 - \leaders 293
 - \left 294
 - \leftghost 844
 - \lefthyphenmin 295
 - \leftmargin kern 675
 - \leftskip 296
 - legacy commands:
 - \legacy_if:n 12214
 - \legacy_if:nTF 111, 12214
 - .legacy_if_gset:n 249, 23044
 - \legacy_if_gset:nn . 111, 12231, 12236
 - \legacy_if_gset_false:n
 - 111, 12223, 12229, 12238
 - .legacy_if_gset_inverse:n 249, 23044
 - \legacy_if_gset_true:n
 - 111, 12223, 12227, 12238
 - \legacy_if_p:n 111, 12214
 - .legacy_if_set:n 249, 23044
 - \legacy_if_set:nn .. 111, 12231, 12231
 - \legacy_if_set_false:n
 - 111, 12223, 12225, 12233
 - .legacy_if_set_inverse:n . 249, 23044
 - \legacy_if_set_true:n
 - 111, 12223, 12223, 12233
 - \legno 297
 - \let 5, 140, 141, 298
 - \lcharcode 845
 - \letterspacefont 676
 - \limits 1497, 299
 - \LineBreak 41,
 - 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 62
 - \linedir 846
 - \linedirection 847
 - \linepenalty 300
 - \lineskip 301
 - \lineskiplimit 302
 - \linewidth 37505
 - \ln 28876, 28879
 - ln 282
 - \localbrokenpenalty 848
 - \localinterlinepenalty 849
 - \localleftbox 854
 - \localrightbox 855
 - \loccount 10275, 10528
 - \loctoks 3227, 3228, 3254
 - logb 282
 - \long 143, 303, 20324, 20328
 - \LongText 38, 76
 - \looseness 304
 - \lower 305
 - \lowercase 67, 306
 - \lpcode 677
 - ltx.pdf.object commands:
 - ltx.pdf.object_id 40916
 - ltx.utils 109, 11896
 - ltx.utils.filedump 109, 11980
 - ltx.utils.filemd5sum 109, 12001
 - ltx.utils.filemoddate 109, 12010
 - ltx.utils.filesize 110, 12063
 - lua commands:
 - \lua_escape:n
 - 109, 11835, 11837, 11842, 11843, 11857
 - \lua_load_module:n
 - 109, 11844, 11845, 11868
 - \lua_now:n
 - 108, 109, 8755, 8764, 11836,
 - 11837, 11837, 11838, 11858, 31739
 - \lua_shipout:n 108, 11837, 11840, 11868
 - \lua_shipout_e:n
 - 108, 11837, 11839, 11841, 11868
 - lua internal commands:
 - \l_lua_err_msg_str ... 11844, 11850
 - _lua_escape:n . 11832, 11832, 11842
 - _lua_load_module_p:n . 11847, 12123
 - _lua_now:n 11832, 11833, 11837
 - _lua_shipout:n . 11832, 11834, 11839
 - \luabytecode 850
 - \luabytecodecall 851
 - luacmd 12073
 - \luacopyinputnodes 852
 - \luaedef 853
 - \luaescapestring 856
 - \luafunction 857
 - \luafunctioncall 858
 - \luatexbanner 859
 - \luatexrevision 860
 - \luatexversion 10, 56, 861
- M**
- \mag 307
 - \mark 308
 - \marks 507
 - \mathaccent 309
 - \mathbin 310
 - \mathchar 311, 20323
 - \mathchardef 312
 - \mathchoice 313
 - \mathclose 314
 - \mathcode 315
 - \mathcolor 1497
 - \mathdefaultsmode 862
 - \mathdelimitersmode 863
 - \mathdir 864
 - \mathdirection 865
 - \mathdisplayskipmode 866
 - \mathemptydisplaymode 869

<code>\matheqdirmode</code>	867	<code>\month</code>	329, 1293, 9099
<code>\matheqnogapstep</code>	868	<code>\moveleft</code>	330
<code>\mathflattenmode</code>	870	<code>\moveright</code>	331
<code>\mathinner</code>	316	msg commands:	
<code>\mathitalicsmode</code>	871	<code>\msg_critical:nn</code>	87, 107, 9568
<code>\mathnolimitsmode</code>	872	<code>\msg_critical:nnn</code>	87, 9568
<code>\mathop</code>	317	<code>\msg_critical:nnnn</code>	87, 9568
<code>\mathopen</code>	318	<code>\msg_critical:nnnnn</code>	87, 9568
<code>\mathoption</code>	873	<code>\msg_critical:nnnnnn</code>	87, 9568
<code>\mathord</code>	319	<code>\msg_critical_text:n</code>	85, 9477, 9482, 9571
<code>\mathpenaltiesmode</code>	874	<code>\msg_error:nn</code> 87, 1954, 1969, 4959,	
<code>\mathpunct</code>	320	4993, 5041, 5044, 5515, 5786, 7214,	
<code>\mathrel</code>	321	7298, 8869, 8952, 9576, 9578, 10335,	
<code>\mathrulesfam</code>	875	10585, 31685, 31731, 37792, 41351	
<code>\mathrulesmode</code>	876	<code>\msg_error:nnn</code> 87, 1652, 1707, 1760,	
<code>\mathrulethicknessmode</code>	878	1765, 1954, 1967, 2161, 2273, 2801,	
<code>\mathscriptboxmode</code>	881	3061, 3101, 3553, 4999, 5222, 5614,	
<code>\mathscriptcharmode</code>	882	5627, 5666, 5699, 5813, 6987, 6994,	
<code>\mathscriptsmode</code>	880	7206, 7312, 8972, 9576, 9577, 9692,	
<code>\mathstyle</code>	883	9839, 10341, 10591, 11508, 11872,	
<code>\mathsurround</code>	322	12679, 13770, 14619, 14680, 17162,	
<code>\mathsurroundmode</code>	884	17186, 17190, 17348, 17721, 20753,	
<code>\mathsurroundskip</code>	885	21010, 22592, 22645, 22784, 22846,	
<code>max</code>	282	22864, 23387, 23799, 24026, 24230,	
<code>\maxdeadcycles</code>	323	24630, 30631, 30872, 30887, 30891,	
<code>\maxdepth</code>	324	30907, 30911, 30967, 30971, 31033,	
<code>md5.HEX</code>	11993	31037, 31115, 31143, 31704, 31762,	
<code>\mdfivesum</code>	773	31768, 36531, 37401, 38797, 38866,	
<code>\meaning</code>	325	39243, 39480, 39517, 39626, 39635,	
<code>\medmuskip</code>	326	39669, 39682, 39697, 39702, 39772,	
<code>\message</code>	327	39791, 39804, 39821, 39831, 39839,	
<code>\MessageBreak</code>	60	40191, 40204, 40227, 40464, 40542,	
meta commands:		40604, 40607, 40650, 40759, 40769,	
<code>.meta:n</code>	249, 23052	40855, 41692, 41700, 41801, 41810	
<code>.meta:nn</code>	250, 23054	<code>\msg_error:nnnn</code>	87,
<code>\middle</code>	508	1643, 1683, 1779, 1954, 1954, 1968,	
<code>min</code>	282	1970, 1977, 2118, 2886, 3081, 3114,	
<code>\mkern</code>	328	3416, 3423, 5199, 5262, 5477, 7218,	
<code>mm</code>	287	7234, 8891, 8910, 8926, 9323, 9576,	
mode commands:		9576, 9718, 9841, 10716, 11732,	
<code>\mode_if_horizontal:</code>	8699	11849, 14712, 17205, 19846, 20934,	
<code>\mode_if_horizontal:TF</code>	73, 8699	20952, 22548, 22602, 22664, 22678,	
<code>\mode_if_horizontal_p:</code>	73, 8699	22855, 22896, 23478, 23531, 24626	
<code>\mode_if_inner:</code>	8701	<code>\msg_error:nnnnn</code> ..	87, 2154, 3569,
<code>\mode_if_inner:TF</code>	74, 8701	7619, 7812, 8453, 9576, 9843, 10727,	
<code>\mode_if_inner_p:</code>	74, 8701	13527, 13568, 17656, 23884, 24067,	
<code>\mode_if_math:</code>	8703	30921, 31190, 38838, 41420, 41872	
<code>\mode_if_math:TF</code>	74, 8703	<code>\msg_error:nnnnnn</code>	
<code>\mode_if_math_p:</code>	74, 8703	 87, 90, 2901, 2915, 7417,	
<code>\mode_if_vertical:</code>	8697	7440, 7514, 9576, 13562, 21548, 24093	
<code>\mode_if_vertical:TF</code>	74, 8697, 22047	<code>\msg_error_text:n</code> .	85, 9477, 9480,
<code>\mode_if_vertical_p:</code>	74, 8697	9485, 9487, 9582, 10231, 41766, 41776	
<code>\mode_leave_vertical:</code>	31,		
	2438, 2438, 22040, 38352, 38413, 40634		

[illegible]

- 14956, 14971, 14985, 14991, 15038,
 15083, 15173, 15289, 15471, 15478,
 15650, 23717, 23720, 23723, 23729,
 23735, 23741, 23747, 23753, 24601,
 24760, 24775, 30928, 30934, 30940,
 30946, 30952, 31869, 31875, 31881,
 31887, 31894, 38607, 38614, 40271,
 40281, 40287, 40294, 40300, 40309,
 40315, 40323, 40332, 40338, 40345,
 40354, 40363, 40378, 40384, 40393,
 40399, 40405, 40411, 40795, 40801,
 40807, 40863, 42742, 42751, 42768
 \msg_none:nn 89, 9635
 \msg_none:nnn 89, 9635
 \msg_none:nnnn 89, 9635
 \msg_none:nnnnn 89, 9635
 \msg_note:nn 88, 9586
 \msg_note:nnn 88, 9586
 \msg_note:nnnn 88, 9586
 \msg_note:nnnnn 88, 9586
 \msg_redirect_class:nn 92, 9775, 9775
 \msg_redirect_module:nnn
 92, 9775, 9777
 \msg_redirect_name:nnn 92, 9766, 9766
 \msg_see_documentation_text:n ...
 86, 9513, 9519
 \msg_set:nnn
 84, 9327, 9346, 41466, 41467
 \msg_set:nnnn
 .. 84, 9327, 9339, 9347, 41464, 41465
 \msg_show:nn 90, 9636
 \msg_show:nnn 90, 9636
 \msg_show:nnnn 90, 9636
 \msg_show:nnnnn 90, 9636
 \msg_show:nnnnnn 90,
 981, 1478, 3995, 4012, 7336, 7346,
 9636, 10381, 10625, 11631, 18167,
 19829, 19851, 21470, 23657, 24195,
 31459, 31475, 38563, 40248, 40691
 \msg_show_item:n
 . 90, 9671, 9671, 18182, 19842, 19856
 \msg_show_item:nn
 90, 9671, 9675, 21514, 31486
 \msg_show_item_unbraced:n
 90, 9671, 9673
 \msg_show_item_unbraced:nn
 90, 661, 9671, 9682, 10389,
 10633, 23674, 38582, 40261, 40269
 \msg_term:nn 89, 9623
 \msg_term:nnn 89, 9623
 \msg_term:nnnn 89, 9623
 \msg_term:nnnnn 89, 9623
 \msg_term:nnnnnn 89, 1478, 9623, 38594
 \msg_warning:nn 88, 5505, 9586
 \msg_warning:nnn .. 88, 5421, 5425,
 5467, 5529, 5567, 5586, 9586, 9835
 \msg_warning:nnnn 88, 5129,
 5276, 9586, 9837, 30855, 30996, 31401
 \msg_warning:nnnnn .. 88, 9586, 41393
 \msg_warning:nnnnnn .. 88, 9586, 9806
 \msg_warning_text:n
 85, 9477, 9489, 9610
 msg internal commands:
 __msg_chk_free:nn . 9319, 9329, 42394
 __msg_chk_if_free:nn 9319
 __msg_class_chk_exist:nTF
 .. 9689, 9689, 9704, 9771, 9781, 9786
 \l__msg_class_loop_seq . 645, 9698,
 9790, 9798, 9808, 9809, 9812, 9814
 __msg_class_new:nn 642,
 9524, 9525, 9555, 9568, 9579, 9608,
 9613, 9618, 9623, 9629, 9635, 9636
 \l__msg_class_tl 643,
 645, 9694, 9711, 9724, 9745, 9749,
 9752, 9760, 9799, 9801, 9803, 9817
 \c__msg_coding_error_text_tl ...
 9348, 9853,
 9861, 9887, 9895, 9904, 9911, 9918,
 9928, 9950, 9959, 9966, 9973, 9981,
 9989, 10021, 10030, 10036, 10043,
 10050, 10062, 10086, 10093, 10109,
 10117, 10126, 10136, 42754, 42771
 \c__msg_continue_text_tl . 9348, 9397
 \c__msg_critical_text_tl . 9348, 9573
 \l__msg_current_class_tl
 645, 9694, 9706,
 9744, 9749, 9752, 9760, 9789, 9803
 __msg_expandable_error:n 655
 __msg_expandable_error:nn
 .. 10208, 10211, 10222, 41764, 41774
 __msg_fatal_exit: .. 9555, 9561, 9563
 \c__msg_fatal_text_tl ... 9348, 9560
 \c__msg_help_text_tl 9348, 9407
 \l__msg_hierarchy_seq
 644, 9697, 9727, 9737, 9742
 __msg_info_aux:NNnnnnnn
 9586, 9586, 9610, 9615, 9620
 __msg_interrupt:n .. 9434, 9443, 9452
 __msg_interrupt:Nnnn 9387
 __msg_interrupt:NnnnN
 9387, 9557, 9570, 9581
 __msg_interrupt_more_text:n ...
 635, 9416, 9418, 9441
 __msg_interrupt_text:n
 9416, 9432, 9436

- _msg_interrupt_wrap:nnn 9395, 9405, [9416](#), 9416
- \c_msg_more_text_prefix_tl 9312, 9332, 9343, 9392, 9409
- \l_msg_name_str [9307](#), 9390, 9423, 9427, 9589, 9597, 9601
- \c_msg_no_info_text_tl .. [9348](#), 9399
- _msg_no_more_text:nnnn [9387](#), 9393, 9415
- \c_msg_on_line_text_tl .. [9348](#), 9383
- _msg_redirect:nnn 9775, 9776, 9778, 9779
- _msg_redirect_loop_chk:nnn ... 9775, 9791, 9796, 9817, 9821
- _msg_redirect_loop_list:n 9775, 9813, 9822
- \l_msg_redirect_prop 9696, 9724, 9769, 9772
- \c_msg_return_text_tl [9348](#), 9856, 9864, 9871
- _msg_show:n .. [641](#), [9636](#), 9640, 9642
- _msg_show:nn [9636](#), 9650, 9653, 9655, 9656
- _msg_show:w [9636](#), 9647, 9654
- _msg_show_dot:w ... [9636](#), 9647, 9652
- _msg_show_eval:nnN [9823](#), 9824, 9826, 9827
- _msg_text:n . [9477](#), 9495, 9496, 9499
- _msg_text:nn [9477](#), 9488, 9490, 9492, 9493
- \c_msg_text_prefix_tl [655](#), [9312](#), 9316, 9330, 9341, 9396, 9406, 9594, 9626, 9632, 9639, 10225
- \l_msg_text_str [9307](#), 9389, 9421, 9426, 9588, 9593, 9600
- _msg_tmp:w 10208, 10218
- \l_msg_tmp_tl [9306](#), 9433, 9439, 9566, 9660, 9666
- \c_msg_trouble_text_tl [9348](#)
- _msg_use:nnnnnnn .. 9534, [9699](#), 9699
- _msg_use_code: [643](#), [9699](#), 9707, 9721, 9725, 9750, 9761
- _msg_use_hierarchy:nwwN [9699](#), 9728, 9729, 9735
- _msg_use_none_delimit_by_s-
stop:w [9311](#), 9311, 9730, 10214
- _msg_use_redirect_module:n ... [644](#), [9699](#), 9732, 9740, 9753
- _msg_use_redirect_name:n [9699](#), 9715, 9722
- \mskip 332
- \muexpr 509
- multichoice commands:
 - .multichoice:nn [250](#), [23056](#)
- \multiply 333
- \muskip 334, 20332
- muskip commands:
 - \c_max_muskip [243](#), [22211](#)
 - \muskip_add:Nn [242](#), [22187](#), 22187, 22191, 42168, 42551
 - \muskip_const:Nn [242](#), [22155](#), 22155, 22160, 22211, 22212, 42308, 42555
 - \muskip_eval:n [243](#), 22158, 22199, 22199, 22206, 22210, 42614
 - \muskip_gadd:Nn [242](#), [22187](#), 22189, 22192, 42250, 42552
 - .muskip_gset:N [250](#), [23068](#)
 - \muskip_gset:Nn [242](#), [22177](#), 22179, 22182, 42249, 42550
 - \muskip_gset_eq:NN [242](#), [22183](#), 22185, 22186, 42252
 - \muskip_gsub:Nn [242](#), [22187](#), 22195, 22198, 42251, 42554
 - \muskip_gzero:N [242](#), [22161](#), 22163, 22166, 22170, 42248
 - \muskip_gzero_new:N [242](#), [22167](#), 22169, 22172
 - \muskip_if_exist:N 22173, 22175
 - \muskip_if_exist:Ntf [242](#), 22168, 22170, [22173](#)
 - \muskip_if_exist_p:N [242](#), [22173](#)
 - \muskip_log:N [243](#), [22207](#), 22207, 22208
 - \muskip_log:n [243](#), [22207](#), 22209
 - \muskip_new:N [242](#), [22149](#), 22149, 22154, 22157, 22168, 22170, 22213, 22214, 22215, 22216
 - .muskip_set:N [250](#), [23068](#)
 - \muskip_set:Nn [242](#), [22177](#), 22177, 22181, 42167, 42549
 - \muskip_set_eq:NN [242](#), [22183](#), 22183, 22184, 42170
 - \muskip_show:N [243](#), [22203](#), 22203, 22204
 - \muskip_show:n [243](#), [1006](#), [22205](#), 22205
 - \muskip_sub:Nn [242](#), [22187](#), 22193, 22197, 42169, 42553
 - \muskip_use:N [243](#), 22200, [22201](#), 22201, 22202
 - \muskip_zero:N [242](#), [22161](#), 22161, 22165, 22168, 42166
 - \muskip_zero_new:N [242](#), [22167](#), 22167, 22171
 - \g_tmpa_muskip [244](#), [22213](#)
 - \l_tmpa_muskip [244](#), [22213](#)
 - \g_tmpb_muskip [244](#), [22213](#)
 - \l_tmpb_muskip [244](#), [22213](#)
 - \c_zero_muskip [243](#), 22162, 22164, [22211](#)

<code>\muskipdef</code>	335	<code>\num</code>	266
<code>\mutoglu</code>	510	<code>\number</code>	346
		<code>\numexpr</code>	511
N			
<code>\n</code>	9007, 9009, 9011, 12121	O	
<code>nan</code>	286	<code>\O</code>	33344, 34885, 35767, 36032
<code>nc</code>	287	<code>\o</code>	33344, 34885, 35779, 36033
<code>nd</code>	287	<code>\odelcode</code>	1207
<code>\newbox</code>	883	<code>\odelimiter</code>	1208
<code>\newcatcodetable</code>	31526	<code>\OE</code>	33345, 34886, 35768
<code>\newcount</code>	883	<code>\oe</code>	33345, 34886, 35780
<code>\newdimen</code>	883	<code>\omathaccent</code>	1209
<code>\newlinechar</code>	59, 336	<code>\omathchar</code>	1210
<code>\newluabytecode</code>	19	<code>\omathchardef</code>	1211
<code>\next</code>	36, 73, 81	<code>\omathcode</code>	1212
<code>\NG</code>	33343, 34884, 35766	<code>\omit</code>	347
<code>\ng</code>	33343, 34884, 35778	opacity commands:	
<code>\noalign</code>	337	<code>\opacity_begin:n</code> ...	342, 40822, 40837
<code>\noautospadding</code>	1169	<code>\opacity_end:</code>	342, 40822, 40843
<code>\noautoxspacing</code>	1170	<code>\opacity_fill:n</code> ...	342, 40822, 40827
<code>\noboundary</code>	338	<code>\opacity_fill_begin:n</code>	
<code>\nobreakspace</code>	35746		342, 40822, 40839
<code>\noexpand</code>	60, 75, 78, 84, 339	<code>\opacity_fill_end:</code> .	342, 40822, 40845
<code>\nohrule</code>	886	<code>\opacity_select:n</code> ..	342, 40822, 40822
<code>\noindent</code>	340	<code>\opacity_stroke:n</code> ..	342, 40822, 40832
<code>\nokerns</code>	887	<code>\opacity_stroke_begin:n</code>	
<code>\noligs</code>	888		342, 40822, 40841
<code>\nolimits</code>	341	<code>\opacity_stroke_end:</code>	342, 40822, 40847
<code>\nonscript</code>	342	opacity internal commands:	
<code>\nonstopmode</code>	343	<code>__opacity_backend_fill:n</code>	
<code>\normaldeviate</code>	948		40829, 40840
<code>\normalend</code>	1311, 1312	<code>__opacity_backend_reset:</code>	
<code>\normaleveryjob</code>	1313		40825, 40844
<code>\normalexpanded</code>	1322	<code>__opacity_backend_reset_fill:</code> ..	
<code>\normalhoffset</code>	1325		40830, 40846
<code>\normalinput</code>	1314	<code>__opacity_backend_reset_stroke:</code>	
<code>\normalitaliccorrection</code>	1324, 1326		40835, 40848
<code>\normallanguage</code>	1315	<code>__opacity_backend_select:n</code>	
<code>\normalleft</code>	1330, 1331		40824, 40838
<code>\normalmathop</code>	1316	<code>__opacity_backend_stroke:n</code>	
<code>\normalmiddle</code>	1332		40834, 40842
<code>\normalmonth</code>	1317	<code>__opacity_select:nN</code>	
<code>\normalouter</code>	1318	..	40822, 40838, 40840, 40842, 40861
<code>\normalover</code>	1319	<code>__opacity_select:nNn</code> ...	40822,
<code>\normalright</code>	1333		40824, 40829, 40834, 40849, 40862
<code>\normalshowtokens</code>	1328	<code>\l_opacity_tmp_fp</code>	
<code>\normalunexpanded</code>	1321	..	40821, 40851, 40853, 40854, 40857
<code>\normalvcenter</code>	1320	<code>\openin</code>	348
<code>\normalvoffset</code>	1327	<code>\openout</code>	349
<code>\nospaces</code>	889	<code>\or</code>	350
<code>\notexpanded: <token></code>	216	or commands:	
<code>\novrule</code>	890	<code>\or:</code>	185, 769, 771, 936,
<code>\nulldelimiterspace</code>	344		1077, 1386, 1388, 2076, 2077, 2078,
<code>\nullfont</code>	345		2079, 2080, 2081, 2082, 2083, 2084,

- 3743, 3744, 3943, 3944, 4146, 4519,
 4520, 4521, 4522, 4793, 4794, 4795,
 4796, 4797, 6364, 6417, 7049, 7051,
 7083, 7084, 7085, 7086, 7087, 7088,
 7089, 7090, 7091, 7092, 7093, 7094,
 7095, 7497, 7498, 10862, 10863,
 10864, 10865, 10866, 10867, 10868,
 14111, 14187, 14528, 14529, 14530,
 14531, 14532, 15560, 15561, 18199,
 18850, 18851, 18852, 18853, 18854,
 18855, 18856, 18857, 18858, 18859,
 18860, 18861, 18862, 18863, 18864,
 18865, 18866, 18867, 18868, 18869,
 18870, 18871, 18872, 18873, 18874,
 18883, 18884, 18885, 18886, 18887,
 18888, 18889, 18890, 18891, 18892,
 18893, 18894, 18895, 18896, 18897,
 18898, 18899, 18900, 18901, 18902,
 18903, 18904, 18905, 18906, 18907,
 20034, 20036, 20038, 20039, 20040,
 20042, 20044, 20046, 20047, 20049,
 20051, 20053, 20055, 24290, 24291,
 24292, 24543, 24558, 24559, 24946,
 24947, 24973, 26263, 26264, 26265,
 26301, 27048, 27049, 27050, 27175,
 27262, 27349, 27350, 27351, 27352,
 27353, 27354, 27355, 27356, 27357,
 27439, 27442, 27781, 27782, 27796,
 27797, 27811, 28108, 28344, 28369,
 28375, 28376, 28377, 28378, 28379,
 28538, 28573, 28575, 28583, 28778,
 28828, 28831, 28840, 28957, 28980,
 28981, 29013, 29014, 29018, 29071,
 29072, 29112, 29117, 29127, 29132,
 29142, 29147, 29157, 29162, 29172,
 29177, 29187, 29192, 29746, 29747,
 29792, 29878, 29881, 29893, 29899,
 29946, 29948, 29949, 29959, 29965,
 30045, 30046, 30053, 30099, 30100,
 30107, 30173, 30174, 30368, 31209,
 31210, 31211, 31288, 31289, 31290
 \oradical 1213
 \orieveryjob 1305, 1306
 \oripdfoutput 1308, 1309
 \outer 883, 351
 \output 352
 \outputbox 891
 \outputmode 949
 \outputpenalty 353
 \over 354
 \overfullrule 355
 \overline 356
 \overwithdelims 357
- P**
- \PackageError 67, 75
 page 339
 \pagebottomoffset 892
 pagebox 339
 \pagedepth 358
 \pagedir 893
 \pagedirection 894
 \pagediscards 512
 \pagefilllstretch 359
 \pagefillstretch 360
 \pagefilstretch 361
 \pagefistretch 1171
 \pagegoal 362
 \pageheight 950
 \pageleftoffset 895
 \pagerightoffset 896
 \pageshrink 363
 \pagestretch 364
 \pagetopoffset 897
 \pagetotal 365
 \pagewidth 951
 \paperheight 41159, 41163
 \paperwidth 41160, 41163
 \par .. 16–21, 96, 412, 1428, 366, 22048,
 36617, 36619, 36626, 36681, 36729,
 36734, 36741, 36746, 36753, 36758
 \pardir 898
 \pardirection 899
 \parfillskip 367
 \parindent 368
 \parshape 369
 \parshapedimen 513
 \parshapeindent 514
 \parshapelength 515
 \parskip 370
 \partokencontext 1215
 \partokenname 1216
 \patterns 371
 \pausing 372
 pc 287
 pdf commands:
 \pdf_destination:nn 346, 41126, 41126
 \pdf_destination:nnnn
 346, 41128, 41128
 \pdf_object_id:n 385
 \pdf_object_id_indexed:nn 385
 \pdf_object_if_exist:n 40966
 \pdf_object_if_exist:nTF . 343, 40966
 \pdf_object_if_exist_p:n . 343, 40966
 \pdf_object_new:n
 343, 40893, 40893, 41470, 41474
 \pdf_object_new:nn 41470, 41471

- \pdf_object_new_indexed:nn 344, 40972, 40972
- \pdf_object_ref:n .. 343, 40893, 40905
- \pdf_object_ref_indexed:nn 344, 40972, 40985
- \pdf_object_ref_last: 345, 41068, 41068
- \pdf_object_unnamed_write:nn ... 344, 41062, 41062, 41067
- \pdf_object_write:n 41476
- \pdf_object_write:nn 41470, 41477, 41484
- \pdf_object_write:nnn 343, 40893, 40898, 40904
- \pdf_object_write_indexed:nnnn .. 344, 40972, 40978, 40984
- \pdf_pageobject_ref:n 345, 41069, 41069
- \pdf_pagesize_gset:nn 345, 41124, 41124
- \pdf_uncompress: ... 345, 40884, 40884
- \pdf_version: 345, 41120, 41120
- \pdf_version_compare:Nn .. 345, 41071
- \pdf_version_compare:NnTF 345, 41071, 41109
- \pdf_version_compare_p:Nn 345, 41071
- \pdf_version_gset:n 345, 41105, 41105
- \pdf_version_major: 345, 41120, 41122
- \pdf_version_min_gset:n 345, 41105, 41107
- \pdf_version_minor: 345, 41120, 41123
- pdf internal commands:
 - __pdf_backend_compress_objects:n 40889
 - __pdf_backend_compresslevel:n 40888
 - __pdf_backend_destination:nn . 41127
 - __pdf_backend_destination:nnnn . 41131
 - __pdf_backend_object_id:n 40912, 40992
 - \g_pdf_backend_object_int 40892, 40896, 40976
 - __pdf_backend_object_last: .. 41068
 - __pdf_backend_object_new: 40895, 40974
 - __pdf_backend_object_now:nn . 41064
 - __pdf_backend_object_ref:n 40907, 40987
 - __pdf_backend_object_write:nnn . 40900, 40980, 41479
 - __pdf_backend_pageobject_ref:n . 41070
 - __pdf_backend_pagesize_gset:nn . 41125, 41150, 41162
 - __pdf_backend_version_major: ... 41076, 41084, 41087, 41096, 41099, 41121, 41122
 - __pdf_backend_version_major_gset:n 41116
 - __pdf_backend_version_minor: ... 41077, 41088, 41100, 41121, 41123
 - __pdf_backend_version_minor_gset:n 41117
 - \g_pdf_init_bool .. 40873, 40886, 40902, 40982, 41065, 41114, 41482
 - \c_pdf_object_block_size_int ... 41028, 41049, 41055, 41058
 - __pdf_object_index_split:nn ... 41021, 41036, 41041
 - \g_pdf_object_prop 41469, 41473, 41481
 - __pdf_object_record:nN 40896, 40916, 40951
 - __pdf_object_record:NnN 40996, 41020, 41025
 - __pdf_object_record:nnN 40975, 40996, 41016, 41060, 41060
 - __pdf_object_retrieve:n 40901, 40908, 40913, 40916, 40956, 40968, 41480
 - __pdf_object_retrieve:Nn 40996, 41035, 41039
 - __pdf_object_retrieve:nn 40981, 40988, 40993, 40996, 41031, 41060, 41061
 - __pdf_version_compare_<:w 41071
 - __pdf_version_compare_=:w 41071
 - __pdf_version_compare_>:w 41071
 - __pdf_version_gset:w 41105, 41106, 41110, 41112
- pdf-attr 339
- \pdfadjustinterwordglue 626
- \pdfadjustspacing 628
- \pdfannot 538
- \pdfappendkern 629
- \pdfcatalog 539
- \pdfcolorstack 541
- \pdfcolorstackinit 542
- \pdfcompresslevel 540
- \pdfcopyfont 630
- \pdfcreationdate 631
- \pdfdecimaldigits 543
- \pdfdest 544
- \pdfdestmargin 545
- \pdfdraftmode 632
- \pdfeachlinedepth 633
- \pdfeachlineheight 634
- \pdfelapsedtime 635

<code>\pdfendlink</code>	546	<code>\pdfnombuiltintounicode</code>	583
<code>\pdfendthread</code>	547	<code>\pdfnoligatures</code>	649
<code>\pdfescapehex</code>	636	<code>\pdfnormaldeviate</code>	650
<code>\pdfescapename</code>	637	<code>\pdfobj</code>	584
<code>\pdfescapestring</code>	638	<code>\pdfobjcompresslevel</code>	585
<code>\pdfextension</code>	900	<code>\pdfomitcharset</code>	586
<code>\pdffakepspace</code>	548	<code>\pdfoutline</code>	587
<code>\pdffeedback</code>	901	<code>\pdfoutput</code>	588
<code>\pdffiledump</code>	702	<code>\pdfpageattr</code>	589
<code>\pdffilemoddate</code>	701	<code>\pdfpagebox</code>	590
<code>\pdffilesize</code>	699	<code>\pdfpageheight</code>	651
<code>\pdffirstlineheight</code>	639	<code>\pdfpageref</code>	591
<code>\pdffontattr</code>	549	<code>\pdfpageresources</code>	592
<code>\pdffontexpand</code>	640	<code>\pdfpagesattr</code>	593
<code>\pdffontname</code>	550	<code>\pdfpagewidth</code>	652
<code>\pdffontobjnum</code>	551	<code>\pdfpkmode</code>	653
<code>\pdffontsize</code>	641	<code>\pdfpkresolution</code>	654
<code>\pdfgamma</code>	552	<code>\pdfprependkern</code>	656
<code>\pdfgentounicode</code>	553	<code>\pdfprimitive</code>	655
<code>\pdfglyphptounicode</code>	554	<code>\pdfprotrudechars</code>	657
<code>\pdfhorigin</code>	555	<code>\pdfptexuseunderscore</code>	594
<code>\pdfignoreddimen</code>	642	<code>\pdfpxdimen</code>	658
<code>\pdfimageapplygamma</code>	556	<code>\pdfrandomseed</code>	659
<code>\pdfimagegamma</code>	557	<code>\pdfrefobj</code>	595
<code>\pdfimagehicolor</code>	558	<code>\pdfrefxform</code>	596
<code>\pdfimageresolution</code>	559	<code>\pdfrefximage</code>	597
<code>\pdfincludechars</code>	560	<code>\pdfresettimer</code>	660
<code>\pdfinclusioncopyfonts</code>	561	<code>\pdfrestore</code>	598
<code>\pdfinclusionerrorlevel</code>	562	<code>\pdfretval</code>	599
<code>\pdfinfo</code>	564	<code>\pdfrunninglinkoff</code>	600
<code>\pdfinfoomitdate</code>	565	<code>\pdfrunninglinkon</code>	601
<code>\pdfinsertht</code>	643	<code>\pdfsave</code>	602
<code>\pdfinterwordsoff</code>	566	<code>\pdfsavepos</code>	661
<code>\pdfinterwordspaceon</code>	567	<code>\pdfsetmatrix</code>	603
<code>\pdflastannot</code>	568	<code>\pdfsetrandomseed</code>	662
<code>\pdflastlinedepth</code>	644	<code>\pdfshellescape</code>	663
<code>\pdflastlink</code>	569	<code>\pdfstartlink</code>	604
<code>\pdflastmatch</code>	645	<code>\pdfstartthread</code>	605
<code>\pdflastobj</code>	570	<code>\pdfstrcmp</code>	138, 5, 698
<code>\pdflastxform</code>	571	<code>\pdfsuppressptexinfo</code>	606
<code>\pdflastximage</code>	572	<code>\pdfsuppresswarningdupdest</code>	607
<code>\pdflastximagecolordepth</code>	573	<code>\pdfsuppresswarningdupmap</code>	609
<code>\pdflastximagepages</code>	575	<code>\pdfsuppresswarningpagegroup</code>	611
<code>\pdflastxpos</code>	646	<code>\pdftexbanner</code>	667
<code>\pdflastypos</code>	647	<code>\pdftexrevision</code>	668
<code>\pdflinkmargin</code>	576	<code>\pdftexversion</code>	669
<code>\pdfliteral</code>	577	<code>\pdfthread</code>	613
<code>\pdfmajorversion</code>	580	<code>\pdfthreadmargin</code>	614
<code>\pdfmapfile</code>	578	<code>\pdftracingfonts</code>	664, 1263, 1264
<code>\pdfmapline</code>	579	<code>\pdftrailer</code>	615
<code>\pdfmatch</code>	648	<code>\pdftrailerid</code>	616
<code>\pdfmdfivesum</code>	700	<code>\pdfunescapehex</code>	665
<code>\pdfminorversion</code>	581	<code>\pdfuniformdeviate</code>	666
<code>\pdfnames</code>	582	<code>\pdfuniquestname</code>	617

- \pdfvariable 902
- \pdfvorigin 618
- \pdfxform 619
- \pdfxformname 620
- \pdfximage 621
- \pdfximagebbox 622
- peek commands:
 - \peek_after:Nw 75, 211, 4131, 20547, 20547, 20560, 20585, 20626
 - \peek_analysis_map_break: 214, 4091, 4091, 4092, 4094, 4114
 - \peek_analysis_map_break:n 214, 4091, 4093, 7912
 - \peek_analysis_map_inline:n . 48, 211, 214, 464, 590, 4103, 4103, 7905
 - \peek_catcode:NwTF .. 212, 20681, 39277
 - \peek_catcode_ignore_spaces:NwTF . 41639
 - \peek_catcode_remove:NwTF 212, 20681, 39343
 - \peek_catcode_remove_ignore_spaces:NwTF 41639
 - \peek_charcode:NwTF 212, 215, 216, 20681
 - \peek_charcode_ignore_spaces:NwTF 41639
 - \peek_charcode_remove:NwTF 212, 215, 20681
 - \peek_charcode_remove_ignore_spaces:NwTF 41639
 - \peek_gafter:Nw ... 211, 20547, 20549
 - \peek_meaning:NwTF .. 212, 20681, 20751
 - \peek_meaning_ignore_spaces:NwTF . 41639
 - \peek_meaning_remove:NwTF . 212, 20681
 - \peek_meaning_remove_ignore_spaces:NwTF 41639
 - \peek_N_type:NwTF 213, 20695, 20727, 20732, 20734
 - \peek_regex:NwTF 215, 7849, 7858, 7864, 7865, 7866
 - \peek_regex:nwTF ... 215, 541, 589, 591, 592, 7849, 7849, 7855, 7856, 7857
 - \peek_regex_remove_once:NwTF 215, 7849, 7877, 7883, 7884, 7885, 7886
 - \peek_regex_remove_once:nwTF 215, 591, 7849, 7867, 7873, 7874, 7875, 7876
 - \peek_regex_replace_once:Nn 216, 7941, 7955
 - \peek_regex_replace_once:nn 216, 7941, 7947
 - \peek_regex_replace_once:NnTF ... 216, 7941, 7949, 7951, 7952, 7953, 7954, 7956
 - \peek_regex_replace_once:nnTF ... 216, 563, 567, 589, 594, 7941, 7941, 7943, 7944, 7945, 7946, 7948
 - \peek_remove_filler:n 213, 20572, 20572, 39274, 39341
 - \peek_remove_spaces:n 211, 212, 20556, 20556, 41648, 41651, 41654, 41657, 41660, 41663
 - \g_peek_token 211, 20536, 20550
 - \l_peek_token 211, 214, 481, 485, 951, 953, 954, 1498, 4136, 4137, 4138, 4139, 4225, 4278, 4281, 4328, 20536, 20548, 20565, 20590, 20593, 20604, 20642, 20654, 20674, 20701, 20702, 20703, 20706, 39290, 39297, 39311
- peek internal commands:
 - __peek_execute_branches_catcode: 954, 20648, 20648
 - __peek_execute_branches_catcode_aux: 20648, 20649, 20651, 20652
 - __peek_execute_branches_catcode_auxii:N 20648, 20656, 20662
 - __peek_execute_branches_catcode_auxiii: 20648, 20659, 20672
 - __peek_execute_branches_charcode: 954, 20648, 20650
 - __peek_execute_branches_meaning: 954, 20640, 20640
 - __peek_execute_branches_N_type: .. 20695, 20698, 20730, 20733, 20735
 - __peek_false:w 954, 20540, 20542, 20558, 20569, 20575, 20605, 20621, 20645, 20668, 20678, 20712, 20725
 - __peek_N_type:w . 20695, 20705, 20715
 - __peek_N_type_aux:nnw 20695, 20707, 20720
 - __peek_remove_filler: 20572, 20585, 20588
 - __peek_remove_filler:w 20572, 20574, 20581, 20583, 20607
 - __peek_remove_filler_expand:w .. 20572, 20598, 20602
 - __peek_remove_spaces: 20556, 20560, 20563
 - \l_peek_search_tl 949, 953, 20539, 20614, 20665, 20675
 - \l_peek_search_token 949, 20538, 20613, 20642
 - __peek_tmp:w 20540, 20543, 20554, 20696, 20718
 - __peek_token_generic:NwTF 954, 20628, 20628, 20630, 20631,

- 20632, 20633, 20729, 20733, 20735
- __peek_token_generic_aux:NNTF .
- 20610, 20610, 20629, 20635
- __peek_token_remove_generic:NNTF
- 954, 20628,
- 20634, 20636, 20637, 20638, 20639
- __peek_true:w
- . 954, 20540, 20540, 20620, 20643,
- 20666, 20676, 20710, 20724, 20725
- __peek_true_aux:w
- .. 950, 952, 20540, 20541, 20553,
- 20560, 20561, 20574, 20615, 20629
- __peek_true_remove:w
- 950, 952, 20551,
- 20551, 20566, 20591, 20595, 20635
- __peek_use_none_delimit_by_s_-
- stop:w ... 954, 20546, 20546, 20708
- \penalty 373
- \pi 25260, 25261
- pi 286
- \postbreakpenalty 1172
- \postdisplaypenalty 374
- \postexhyphenchar 903
- \posthyphenchar 904
- \prebinoppenalty 905
- \prebreakpenalty 1173
- \predisplaydirection 516
- \predisplaygapfactor 906
- \predisplaypenalty 375
- \predisplaysize 376
- \preexhyphenchar 907
- \prehyphenchar 908
- \prerelpenalty 909
- \pretolerance 377
- \prevdepth 378
- \prevgraf 379
- prg commands:
- \prg_break:
- 75, 562, 805, 871, 872, 2435,
- 2436, 3472, 3547, 3938, 4023, 4053,
- 4054, 4055, 4056, 4057, 4058, 4418,
- 4684, 4688, 5947, 5957, 5962, 5971,
- 5995, 6040, 6903, 7726, 8714, 13398,
- 14590, 14606, 14726, 14758, 14864,
- 14867, 15008, 15055, 15108, 15114,
- 15348, 15429, 15601, 15742, 16641,
- 16648, 17897, 17930, 17981, 18035,
- 18050, 18057, 18645, 19198, 19676,
- 24352, 24361, 26387, 26407, 26408,
- 26696, 26697, 26710, 26800, 26801,
- 26802, 30283, 30309, 30536, 41264
- \prg_break:n
- 75, 2435, 2437, 6440, 6931,
- 8714, 13400, 14492, 14500, 14512,
- 17771, 17910, 18655, 19104, 24151,
- 24170, 24184, 24368, 30796, 30807
- \prg_break_point:
- 75, 451, 458, 860, 2435,
- 2435, 2436, 2437, 3299, 3341, 3465,
- 3472, 3855, 4024, 4060, 4414, 4662,
- 5943, 5992, 6441, 6773, 6925, 7720,
- 7726, 8714, 13388, 14493, 14501,
- 14591, 14607, 14727, 14759, 14865,
- 14868, 15009, 15056, 15109, 15115,
- 15349, 15549, 15706, 16642, 17768,
- 17898, 17932, 17983, 18035, 18051,
- 18103, 18110, 18650, 19098, 19198,
- 19676, 24145, 24164, 24179, 24353,
- 24362, 26388, 26409, 26698, 26805,
- 30284, 30309, 30544, 30797, 41295
- \prg_break_point:Nn 74, 153,
- 423, 481, 492, 872, 894, 992, 2426,
- 2426, 2427, 3985, 4114, 6665, 6679,
- 6724, 7911, 8714, 10477, 10496,
- 12933, 12963, 12974, 13935, 13961,
- 13981, 14003, 17933, 17974, 17984,
- 18007, 18015, 18024, 18052, 18700,
- 19548, 19570, 19593, 19620, 19642,
- 21385, 21407, 21423, 21887, 22445,
- 26879, 33442, 34941, 35489, 35499
- \prg_do_nothing:
- 14, 75,
- 511, 565, 585, 695, 720, 786, 853,
- 859, 910, 927, 1084, 1265, 2424,
- 2424, 2435, 2844, 2871, 2970, 2971,
- 2972, 3387, 3545, 3546, 3826, 3875,
- 4179, 4510, 4995, 5038, 5039, 5046,
- 5047, 6985, 7213, 7636, 7640, 7692,
- 8984, 10861, 11158, 11575, 11614,
- 11616, 12500, 13282, 13755, 13757,
- 14656, 15599, 17285, 17321, 17322,
- 17459, 17466, 17863, 17865, 19191,
- 19197, 19205, 19367, 19568, 19578,
- 19641, 19652, 19730, 19742, 19797,
- 19801, 19808, 23484, 24636, 24672,
- 24702, 24710, 26272, 30264, 30666,
- 30820, 31691, 33485, 33496, 41223
- \prg_generate_conditional_-
- variant:Nnn
- .. 35, 67, 3050, 3050, 7364, 7370,
- 7400, 7402, 8423, 10298, 11147,
- 11297, 11402, 11406, 11410, 11414,
- 11439, 11450, 11491, 12784, 12794,
- 12818, 12821, 12837, 12851, 12857,
- 12868, 12888, 12919, 12926, 13183,
- 13201, 13210, 13834, 13846, 13854,
- 13885, 13920, 17128, 17150, 17631,
- 17632, 17712, 17772, 17866, 17868,
- 17882, 17884, 17886, 17888, 19134,

- 19385, 19399, 19400, 19537, 19539,
21076, 21078, 21080, 21217, 21219,
21352, 21375, 23626, 36481, 36483,
36487, 37394, 41507, 41509, 41590
\prg_gset_conditional:Nnn
..... 66, 1616, 1618
\prg_gset_conditional:Npnn
..... 66, 1595, 1597,
1824, 1838, 1851, 1866, 1880, 1895
\prg_gset_eq_conditional:NNn ...
..... 67, 1739, 1741
\prg_gset_protected_conditional:Nnn
..... 66, 1616, 1624
\prg_gset_protected_conditional:Npnn
..... 66, 1595, 1603
\prg_map_break:Nn
..... 74, 423, 480, 736, 923,
979, 1014, 2426, 2427, 2433, 4092,
4094, 4409, 8714, 10460, 10462,
13001, 13003, 13994, 13996, 17921,
17923, 19655, 19657, 21439, 21441,
22447, 22448, 33456, 35145, 35147
\prg_new_conditional:Nnn
..... 66, 1616, 1620, 8367
\prg_new_conditional:Npnn
..... 66, 67, 387, 742, 939, 1595,
1599, 2250, 4804, 4833, 4916, 4945,
8367, 8415, 8466, 8527, 8542, 8553,
8568, 8578, 8697, 8699, 8701, 8703,
9314, 10394, 11442, 11485, 12214,
12776, 12786, 12801, 12810, 12872,
12889, 12900, 13171, 13185, 13203,
13244, 13264, 13279, 13817, 13824,
13829, 13836, 13841, 14489, 14498,
14513, 14521, 15195, 15229, 15248,
16631, 16638, 17112, 17120, 17130,
17140, 17290, 17301, 17704, 18431,
18484, 18516, 18554, 18562, 19126,
19207, 19490, 20163, 20168, 20173,
20178, 20185, 20191, 20197, 20202,
20207, 20212, 20217, 20224, 20229,
20236, 20251, 20256, 20292, 20336,
20351, 20428, 20437, 21334, 21354,
21679, 21708, 22105, 22114, 23620,
23635, 24538, 25703, 26589, 26597,
26613, 32808, 32863, 32872, 34241,
34270, 34303, 34381, 34399, 34417,
34442, 35131, 36477, 36479, 36485,
37384, 38647, 40966, 41071, 41963
\prg_new_eq_conditional:NNn
.. 67, 1739, 1743, 8367, 8462, 8464,
12295, 12296, 12820, 13806, 13808,
13810, 13812, 13814, 17541, 17543,
18161, 18162, 18163, 18164, 18165,
18166, 18356, 18358, 19122, 19124,
19297, 19299, 19486, 19488, 20222,
21330, 21332, 21609, 21611, 22079,
22081, 22173, 22175, 24191, 24193,
26587, 26588, 31297, 31299, 31326,
31328, 31750, 31752, 36422, 36424
\prg_new_protected_conditional:Nnn
..... 66, 1616, 1626, 8367
\prg_new_protected_conditional:Npnn
..... 66, 953, 1595, 1605, 4525,
7359, 7365, 7395, 7397, 8367, 8962,
10289, 10414, 10434, 11136, 11289,
11400, 11404, 11408, 11412, 11430,
12825, 12838, 12859, 12914, 12921,
13848, 13856, 14630, 14639, 17627,
17629, 17753, 17862, 17864, 17870,
17873, 17876, 17879, 19376, 19386,
19388, 19504, 19508, 21071, 21205,
21211, 30786, 31362, 31754, 40706
\prg_replicate:nn
73, 122, 164, 588, 613, 1058, 1330,
4369, 5001, 5754, 6371, 6397, 6543,
6551, 6714, 6879, 6990, 7651, 7659,
7706, 7824, 7826, 8649, 8649, 9424,
9598, 10692, 16714, 16721, 16730,
16740, 16753, 16980, 17006, 24100,
28047, 28928, 29237, 29493, 29539,
29576, 30130, 30138, 31129, 31232,
31347, 32356, 32367, 32372, 39853,
39861, 39929, 39963, 39975, 39978,
40058, 40059, 41208, 41214, 41339
\prg_return_false:
..... 66, 67, 400, 580, 867, 887,
919, 972, 976, 1589, 1591, 1663,
1671, 1833, 1844, 1860, 1875, 1888,
1904, 2253, 4530, 4830, 4856, 4921,
4950, 7477, 8367, 8420, 8471, 8532,
8548, 8558, 8574, 8584, 8698, 8700,
8702, 8704, 8966, 8974, 9317, 10296,
10401, 10417, 10437, 11145, 11294,
11421, 11436, 11459, 11468, 11479,
11488, 12218, 12781, 12791, 12806,
12815, 12834, 12848, 12865, 12878,
12896, 12911, 12917, 12924, 13180,
13198, 13218, 13226, 13236, 13252,
13275, 13286, 13822, 13827, 13832,
13839, 13844, 13852, 13860, 14494,
14502, 14518, 14534, 14637, 14646,
15199, 15202, 15205, 15232, 15235,
15252, 15255, 15258, 16634, 16641,
17117, 17125, 17136, 17146, 17295,
17306, 17659, 17709, 17767, 17786,
18429, 18461, 18466, 18489, 18521,
18559, 18567, 19131, 19223, 19226,

- 19379, 19393, 19493, 19528, 19534,
20166, 20171, 20176, 20181, 20188,
20195, 20200, 20205, 20210, 20215,
20220, 20227, 20232, 20249, 20254,
20259, 20264, 20298, 20301, 20313,
20347, 20360, 20441, 20466, 20483,
20492, 21074, 21209, 21215, 21349,
21363, 21367, 21682, 21725, 21740,
21741, 22109, 22117, 23632, 23653,
24549, 24551, 25716, 25726, 26594,
26608, 26621, 30796, 31372, 31764,
31770, 32818, 32821, 32867, 32877,
34249, 34253, 34259, 34295, 34315,
34363, 34395, 34413, 34436, 34458,
35140, 36478, 36480, 36486, 37390,
37392, 38652, 38655, 40730, 40969,
41079, 41091, 41103, 41968, 42736
- \prg_return_true:
..... 66, 67, 400, 576, 580, 684,
730, 742, 867, 972, 976, 1589, 1589,
1663, 1671, 1831, 1842, 1858, 1873,
1886, 1902, 2253, 4528, 4828, 4854,
4919, 4948, 7475, 8367, 8418, 8469,
8530, 8546, 8556, 8572, 8582, 8698,
8700, 8702, 8704, 8987, 9317, 10294,
10399, 10404, 10407, 10420, 10440,
11143, 11295, 11422, 11437, 11457,
11466, 11477, 11489, 12220, 12779,
12789, 12804, 12813, 12832, 12846,
12865, 12876, 12894, 12909, 12917,
12924, 13178, 13196, 13216, 13234,
13250, 13273, 13284, 13822, 13827,
13832, 13839, 13844, 13852, 13860,
14512, 14516, 14524, 14537, 14637,
14646, 15199, 15205, 15237, 15252,
15258, 16634, 16647, 17115, 17123,
17134, 17144, 17293, 17304, 17670,
17707, 17771, 17789, 18461, 18487,
18519, 18557, 18565, 19129, 19219,
19222, 19228, 19382, 19396, 19494,
19524, 19534, 20166, 20171, 20176,
20181, 20188, 20195, 20200, 20205,
20210, 20215, 20220, 20227, 20232,
20248, 20254, 20262, 20312, 20345,
20358, 20464, 20490, 21074, 21209,
21215, 21337, 21347, 21367, 21373,
21682, 21741, 22108, 22118, 23631,
23652, 24542, 24547, 25711, 25732,
26592, 26610, 26619, 30790, 30793,
30807, 31370, 31760, 32819, 32867,
32877, 34257, 34262, 34266, 34278,
34281, 34284, 34287, 34290, 34293,
34313, 34319, 34322, 34325, 34328,
34331, 34334, 34337, 34340, 34343,
34346, 34349, 34352, 34355, 34358,
34361, 34390, 34393, 34408, 34411,
34425, 34428, 34431, 34434, 34450,
34453, 34456, 35139, 36478, 36480,
36486, 37389, 38653, 40731, 40970,
41078, 41090, 41102, 41967, 42737
- \prg_set_conditional:Nnn
..... 66, 1616, 1616, 8367
- \prg_set_conditional:Npnn
..... 66, 67, 406, 1595, 1595, 8367, 42730
- \prg_set_eq_conditional:NNn
..... 67, 1739, 1739, 8367
- \prg_set_protected_conditional:Nnn
..... 66, 1616, 1622, 8367
- \prg_set_protected_conditional:Npnn
..... 66, 1595, 1601, 8367
- prg internal commands:
- __prg_break_point:Nn 422
- __prg_F_true:w 1692, 1725, 1737
- __prg_generate_conditional:nnNNNnnn
..... 1611, 1640, 1649, 1649
- __prg_generate_conditional:NNnnnnNw
..... 1649, 1658, 1675, 1690
- __prg_generate_conditional_
count:NNNnn 1616, 1617,
1619, 1621, 1623, 1625, 1627, 1628
- __prg_generate_conditional_
count:nnNNNnn 1616, 1632, 1637
- __prg_generate_conditional_
fast:nw . 400, 401, 1649, 1662, 1673
- __prg_generate_conditional_
parm:NNNpnn 1595, 1596,
1598, 1600, 1602, 1604, 1606, 1607
- __prg_generate_conditional_
test:w 1649, 1660, 1670
- __prg_generate_F_form:wNNnnnnN .
..... 1692, 1719
- __prg_generate_p_form:wNNnnnnN .
..... 400, 1692, 1692
- __prg_generate_T_form:wNNnnnnN .
..... 1692, 1711
- __prg_generate_TF_form:wNNnnnnN
..... 1692, 1727
- __prg_p_true:w 1692, 1704, 1735
- __prg_replicate:N
..... 8649, 8656, 8657, 8659
- __prg_replicate_ 8649
- __prg_replicate_0:n 8649
- __prg_replicate_1:n 8649
- __prg_replicate_2:n 8649
- __prg_replicate_3:n 8649
- __prg_replicate_4:n 8649
- __prg_replicate_5:n 8649
- __prg_replicate_6:n 8649

- __prg_replicate_7:n [8649](#)
- __prg_replicate_8:n [8649](#)
- __prg_replicate_9:n [8649](#)
- __prg_replicate_first:N
..... [8649](#), [8652](#), [8658](#)
- __prg_replicate_first_0:n ... [8649](#)
- __prg_replicate_first_1:n ... [8649](#)
- __prg_replicate_first_2:n ... [8649](#)
- __prg_replicate_first_3:n ... [8649](#)
- __prg_replicate_first_4:n ... [8649](#)
- __prg_replicate_first_5:n ... [8649](#)
- __prg_replicate_first_6:n ... [8649](#)
- __prg_replicate_first_7:n ... [8649](#)
- __prg_replicate_first_8:n ... [8649](#)
- __prg_replicate_first_9:n ... [8649](#)
- __prg_set_eq_conditional:NNNn ..
..... [1739](#), [1740](#), [1742](#), [1744](#), [1745](#)
- __prg_set_eq_conditional:nnNnnNNw
..... [1749](#), [1757](#), [1757](#)
- __prg_set_eq_conditional_F_-
form:nnn [1757](#)
- __prg_set_eq_conditional_F_-
form:wNnnnn [1794](#), [42411](#)
- __prg_set_eq_conditional_-
loop:nnnnNw . [1757](#), [1769](#), [1771](#), [1786](#)
- __prg_set_eq_conditional_p_-
form:nnn [1757](#)
- __prg_set_eq_conditional_p_-
form:wNnnnn [1788](#), [42399](#)
- __prg_set_eq_conditional_T_-
form:nnn [1757](#)
- __prg_set_eq_conditional_T_-
form:wNnnnn [1792](#), [42407](#)
- __prg_set_eq_conditional_TF_-
form:nnn [1757](#)
- __prg_set_eq_conditional_TF_-
form:wNnnnn [1790](#), [42403](#)
- __prg_T_true:w [1692](#), [1717](#), [1736](#)
- __prg_TF_true:w [402](#), [1692](#), [1733](#), [1738](#)
- __prg_use_none_delimit_by_q_-
recursion_stop:w
.. [1593](#), [1593](#), [1678](#), [1762](#), [1767](#), [1774](#)
- \primitive [775](#)
- prop commands:
- \c_empty_prop [228](#), [960](#), [961](#),
[976](#), [20803](#), [20807](#), [20841](#), [21060](#), [21336](#)
- \prop_clear:N
[219](#), [220](#), [966](#), [20832](#), [20832](#), [20834](#),
[20865](#), [20871](#), [38185](#), [39188](#), [42171](#)
- \prop_clear_new:N [220](#), [20864](#),
[20864](#), [20866](#), [39393](#), [39425](#), [39466](#)
- \prop_clear_new_linked:N
..... [220](#), [20864](#), [20870](#), [20872](#)
- \prop_concat:NNN [219](#), [222](#),
[223](#), [20968](#), [20968](#), [20970](#), [42125](#), [42172](#)
- \prop_const_from_keyval:Nn
..... [221](#), [21023](#), [21023](#), [21028](#),
[36050](#), [37355](#), [37362](#), [40171](#), [42309](#)
- \prop_const_linked_from_keyval:Nn
.... [221](#), [21023](#), [21029](#), [21034](#), [42310](#)
- \prop_count:N [224](#), [21442](#), [21442](#), [21451](#)
- \prop_gclear:N [220](#), [20832](#),
[20835](#), [20837](#), [20868](#), [20874](#), [42253](#)
- \prop_gclear_new:N [220](#), [1449](#),
[20864](#), [20867](#), [20869](#), [37427](#), [37428](#)
- \prop_gclear_new_linked:N
..... [220](#), [20864](#), [20873](#), [20875](#)
- \prop_gconcat:NNN
[222](#), [20968](#), [20971](#), [20973](#), [42126](#), [42254](#)
- \prop_get:NnN [151](#), [152](#),
[219](#), [223](#), [224](#), [968](#), [21063](#), [21063](#),
[21068](#), [21069](#), [21070](#), [21071](#), [21076](#),
[21078](#), [21080](#), [38431](#), [38435](#), [38505](#),
[38509](#), [38662](#), [38777](#), [38877](#), [39968](#)
- \prop_get:NnNTF .. [223](#), [225](#), [9724](#),
[9744](#), [9799](#), [10373](#), [10617](#), [14699](#),
[21063](#), [22904](#), [31622](#), [38771](#), [38882](#),
[39403](#), [39662](#), [39675](#), [39690](#), [39769](#),
[39797](#), [39813](#), [40184](#), [40197](#), [40627](#)
- \prop_gpop:NnN . [223](#), [21189](#), [21195](#),
[21203](#), [21204](#), [21211](#), [21219](#), [42255](#)
- \prop_gpop:NnNTF
[223](#), [226](#), [21189](#), [42256](#), [42257](#), [42258](#)
- .prop_gput:N [250](#), [23076](#)
- \prop_gput:Nnn [222](#), [966](#), [3599](#), [3600](#),
[8325](#), [8326](#), [9506](#), [10245](#), [10246](#),
[10253](#), [10254](#), [10256](#), [10257](#), [10258](#),
[10259](#), [10279](#), [10325](#), [10532](#), [10576](#),
[11892](#), [11893](#), [14460](#), [14461](#), [14462](#),
[14463](#), [14464](#), [14465](#), [14466](#), [14467](#),
[14468](#), [14469](#), [14470](#), [14471](#), [14472](#),
[14473](#), [14474](#), [14481](#), [14484](#), [15336](#),
[15337](#), [21221](#), [21223](#), [21243](#), [21248](#),
[21250](#), [21255](#), [22394](#), [22395](#), [23766](#),
[23767](#), [24609](#), [24610](#), [31398](#), [31505](#),
[31506](#), [31569](#), [31910](#), [31911](#), [37652](#),
[37670](#), [37691](#), [37709](#), [37740](#), [39587](#),
[39588](#), [39589](#), [39590](#), [39591](#), [39592](#),
[39593](#), [39594](#), [39605](#), [39607](#), [39609](#),
[39610](#), [39611](#), [39612](#), [39613](#), [39614](#),
[39615](#), [39715](#), [39716](#), [39730](#), [39744](#),
[39754](#), [40417](#), [40418](#), [41473](#), [42259](#)
- \prop_gput_from_keyval:Nn
..... [223](#), [966](#), [20991](#),
[20994](#), [20996](#), [21020](#), [31323](#), [42261](#)
- \prop_gput_if_new:Nnn
..... [41668](#), [41671](#), [41674](#)

- \prop_gput_if_not_in:Nnn
 222, 21221,
 21227, 21264, 41670, 41671, 42260
- \prop_gremove:Nn 224, 10358, 10602,
 21175, 21181, 21188, 31567, 42262
- \prop_gset_eq:NN
 . 220, 20876, 20879, 20881, 37429,
 37431, 37596, 37598, 37643, 37645,
 37897, 38063, 38104, 42108, 42263
- \prop_gset_from_keyval:Nn
 221, 21011, 21017, 21022, 31315, 42264
- \prop_if_empty:N 21334, 21352
- \prop_if_empty:NTF
 225, 21334, 21455, 38651
- \prop_if_empty_p:N 225, 21334
- \prop_if_exist:N 21330, 21332
- \prop_if_exist:NTF
 224, 20796, 20865, 20868,
 20871, 20874, 21330, 22803, 38649
- \prop_if_exist_p:N 224, 21330
- \prop_if_in:Nn 219, 21354, 21375
- \prop_if_in:NnTF 225,
 9509, 9515, 21354, 31387, 31445, 39405
- \prop_if_in_p:Nn 225, 21354
- \prop_item:Nn
 . 219, 224, 226, 969, 9510, 9516,
 21104, 21104, 21121, 31391, 31450,
 36089, 37621, 40031, 40070, 41481
- \prop_log:N .. 228, 21470, 21472, 21473
- \prop_make_flat:N
 219, 221, 20926, 20926, 20935, 20938
- \prop_make_linked:N ... 219, 221,
 960, 20944, 20944, 20953, 20956, 36079
- \prop_map_break: ... 227, 977, 978,
 21381, 21382, 21383, 21384, 21385,
 21407, 21419, 21420, 21421, 21422,
 21423, 21438, 21438, 21439, 21441
- \prop_map_break:n
 227, 21119, 21373, 21438, 21440
- \prop_map_function:NN
 90, 226, 978, 10388,
 10632, 21377, 21377, 21399, 21447,
 21463, 21514, 31486, 38580, 40259
- \prop_map_inline:Nn
 226, 21400, 21400, 21414,
 37907, 37909, 37912, 37930, 37932,
 38006, 38023, 38084, 38086, 38090,
 38092, 38272, 38291, 38479, 38488
- \prop_map_tokens:Nn 226, 873, 970,
 976, 21108, 21358, 21415, 21415, 21437
- \prop_new:N 219–
 221, 9504, 9505, 9527, 9696, 10267,
 10520, 14459, 20804, 20804, 20809,
 20865, 20868, 20964, 20965, 20966,
 20967, 21025, 22486, 22487, 22803,
 31308, 31314, 31516, 37885, 37886,
 37887, 38361, 38402, 39491, 39581,
 39586, 39603, 39608, 40593, 41469
- \prop_new_linked:N 219–221, 20810,
 20810, 20831, 20871, 20874, 21031
- \prop_pop:NnN 219, 223, 21189, 21189,
 21201, 21202, 21205, 21217, 42173
- \prop_pop:NnNTF 223,
 225, 967, 21189, 42174, 42175, 42176
- .prop_put:N 250, 23076
- \prop_put:Nnn 219, 222,
 223, 436, 957, 965, 973, 9772, 9788,
 9805, 21221, 21221, 21229, 21234,
 21236, 21241, 22915, 37649, 37667,
 37686, 37707, 37738, 37941, 37943,
 37949, 37951, 37960, 37966, 37974,
 38033, 38041, 38131, 38137, 38145,
 38152, 38296, 38362, 38364, 38366,
 38368, 38370, 38372, 38374, 38376,
 38378, 38380, 38382, 38384, 38386,
 38388, 38390, 38392, 38394, 38396,
 38749, 39189, 39394, 39413, 39456,
 39471, 39494, 39652, 40594, 42177
- \prop_put_from_keyval:Nn ... 223,
 966, 20991, 20991, 20993, 21014, 42179
- \prop_put_if_new:Nnn
 41668, 41669, 41672
- \prop_put_if_not_in:Nnn
 222, 965, 21221,
 21225, 21257, 41668, 41669, 42178
- \prop_remove:Nn
 219, 224, 9769, 9784, 21175, 21175,
 21187, 38473, 38477, 38481, 42180
- \prop_set_eq:NN 219, 220, 961, 964,
 965, 20876, 20876, 20878, 37584,
 37586, 37636, 37638, 37894, 37903,
 37905, 38056, 38080, 38082, 38101,
 38229, 38468, 39476, 42107, 42181
- \prop_set_from_keyval:Nn ... 221,
 223, 21011, 21011, 21016, 39631, 42182
- \prop_show:N 227,
 967, 980, 21059, 21470, 21470, 21471
- \prop_to_keyval:N .. 224, 21452, 21452
- \g_tmpa_prop 228, 20964
- \l_tmpa_prop 228, 20964
- \g_tmpb_prop 228, 20964
- \l_tmpb_prop 228, 20964
- prop internal commands:
 \c__prop_basis_int
 959, 20775, 20785, 20788, 20790
- __prop_chk:w 956, 958, 960,
 963, 967, 977, 20748, 20748, 20765,
 20803, 21053, 21126, 21282, 21480

```

\__prop_chk_get:nw 20748, 20752, 20755
\__prop_chk_loop:nw .....
..... 20748, 20748, 20749, 20756
\__prop_clear:NNN ..... 20832,
20833, 20836, 20838, 21013, 21019
\__prop_clear:wNNN .....
..... 961, 963, 20832, 20842, 20844, 20941
\__prop_clear_entries:NN .....
..... 961, 20847, 20851, 20898
\__prop_clear_loop:Nw .....
..... 961, 20832, 20853, 20856, 20861
\__prop_concat:nNNN .....
..... 20968, 20977, 20980, 20983
\__prop_concat:NNNNN .....
..... 20968, 20969, 20972, 20974
\__prop_count:nn . 21442, 21447, 21450
\__prop_flatten:N .....
..... 959, 20760, 20760, 20890
\__prop_flatten:w .....
..... 957, 960, 980, 981, 20758,
20758, 20799, 20802, 20820, 20845,
20848, 20896, 20906, 21090, 21137,
21290, 21340, 21344, 21505, 21518
\__prop_flatten_aux:N .....
..... 959, 20759, 20760, 20762, 20763
\__prop_flatten_aux:w .....
..... 20760, 20761, 20762
\__prop_flatten_loop:w .....
..... 959, 20760, 20766, 20769, 20773
\__prop_from_keyval:nn ... 20991,
20992, 20995, 20997, 21026, 21032
\__prop_from_keyval:Nnn .....
..... 20991, 21000, 21001, 21003
\__prop_get:NnnTF .....
..... 968, 21063, 21065, 21073, 21082
\__prop_get_linked:w ..... 969,
976, 21063, 21087, 21089, 21114, 21366
\__prop_get_linked_aux:w .....
..... 969, 21063, 21093, 21096, 21102
\__prop_if_empty:w ..... 976
\__prop_if_empty_return:w .....
..... 21334, 21339, 21343
\__prop_if_flat:NTF .. 980, 20794,
20794, 20840, 20886, 20888, 20930,
20948, 21106, 21356, 21478, 21489
\__prop_if_flat_aux:w .....
..... 20794, 20797, 20801
\__prop_if_in_flat:nnn .....
..... 21354, 21360, 21370
\__prop_if_recursion_tail_stop:n
..... 20745, 20745, 20746, 21499
\__prop_item:nnn .....
..... 970, 21104, 21110, 21116
\prop_make_flat:N\__prop_make_-
flat:Nn ..... 20926
\__prop_make_flat:Nn .. 20931, 20939
\__prop_make_linked:Nn .....
..... 20944, 20949, 20957
\__prop_map_function:Nw .....
..... 977, 21377, 21380, 21387, 21397
\__prop_map_tokens:nw .....
..... 21415, 21418, 21425, 21435
\__prop_missing_eq:n .....
..... 965, 20991, 21006, 21009
\__prop_new_linked:N .....
..... 964, 20810, 20813, 20815, 20959
\__prop_next_prefix: .....
..... 20777, 20777, 20817
\__prop_pair:wn .....
..... 956-958, 967, 978, 982, 983, 20751,
20755, 20757, 20757, 20767, 20769,
20772, 20854, 20857, 20909, 20912,
20918, 20985, 20986, 20989, 21043,
21046, 21275, 21381, 21382, 21383,
21384, 21388, 21389, 21390, 21391,
21403, 21405, 21410, 21419, 21420,
21421, 21422, 21426, 21427, 21428,
21429, 21483, 21497, 21500, 21571
\__prop_pop:NnNNnTF .....
..... 971, 21122, 21122, 21177,
21183, 21191, 21197, 21207, 21213
\__prop_pop_linked:NNNn .....
..... 21122, 21140, 21148
\__prop_pop_linked:w .....
..... 971, 21122, 21153, 21156
\__prop_pop_linked:wnNNnTF .....
..... 970, 971, 21122, 21132, 21136
\__prop_pop_linked_next:w .....
..... 21122, 21168, 21174
\__prop_pop_linked_prev:w .....
..... 21122, 21162, 21172, 21309
\g__prop_prefix_int .....
..... 959, 20775, 20780, 20781
\__prop_put:NNNnn ..... 21221
\__prop_put:nNNnn .....
..... 20987, 20992, 20995, 21026, 21032,
21222, 21224, 21226, 21228, 21271
\__prop_put_linked:NNNN ..... 21221
\__prop_put_linked:NNnN 21292, 21296
\__prop_put_linked:wnNN .. 974, 21221
\__prop_put_linked:wnnN 21287, 21289
\__prop_put_linked_new:w .....
..... 974, 975, 21221, 21299, 21304
\__prop_put_linked_old:w .....
..... 974, 975, 21221, 21301, 21320
\__prop_set_eq:NNNN .....
.. 20876, 20877, 20880, 20882, 20979

```

- __prop_set_eq:nNnNN 963, 20876, 20899, 20901
 - __prop_set_eq:wNNNN 964, 20876, 20892, 20895, 20961
 - __prop_set_eq_end:w 963, 20876, 20916, 20920
 - __prop_set_eq_loop:NNnw 20876, 20905, 20911, 20917, 20921
 - __prop_show:NN 21470, 21470, 21472, 21474
 - __prop_show_bad_name:NNN 981, 982, 21470, 21534, 21546
 - __prop_show_end:NNN 981, 21470, 21533, 21554
 - __prop_show_finally:NNn 981, 21470, 21490, 21510, 21560
 - __prop_show_flat:w 980, 982, 21470, 21481, 21497, 21501, 21570
 - __prop_show_linked:w 980, 21470, 21486, 21503
 - __prop_show_loop:NNw 982, 21470, 21537, 21539, 21575
 - __prop_show_loop_key:wNNN 981, 982, 21470, 21528, 21564
 - __prop_show_prepare:w 981, 21470, 21493, 21517
 - __prop_split:NnTFn 967, 968, 974, 976, 21035, 21035, 21084, 21124, 21278
 - __prop_split_aux:nNTFn 21035, 21037, 21040
 - __prop_split_flat:w 967, 968, 21035, 21042, 21050, 21053
 - __prop_split_linked:w 967, 21035, 21042, 21054, 21056
 - __prop_split_test:wn 967, 21035, 21045, 21051
 - __prop_split_wrong:Nw 967, 968, 21035, 21046, 21057
 - __prop_tmp:w ... 981, 982, 20739, 20739, 20985, 20989, 21522, 21541
 - \l__prop_tmp_tl 959, 960, 963, 964, 974, 975, 981, 982, 20740, 20779, 20821, 20822, 20823, 20825, 20914, 20915, 20917, 20924, 20960, 20962, 21273, 21283, 21286, 21315, 21326, 21492, 21549, 21557, 21561, 21567
 - __prop_to_keyval:nn 21452, 21463, 21468
 - __prop_to_keyval:nnw 21452
 - __prop_to_keyval_exp_after:wN 21452
 - __prop_to_prefix:n 20777, 20780, 20783, 20787, 20789
 - prop <prefix> internal commands:
 - __prop <prefix> 956, 957
 - \protect 1350, 1405, 10759, 25195, 32968, 33207, 33222, 33231, 33245, 33247, 35681
 - \protected . 82, 84, 106, 517, 20326, 20328
 - \protrudechars 952
 - \protrusionboundary 910
 - \ProvidesExplClass 10
 - \ProvidesExplFile 10, 41679
 - \ProvidesExplPackage 10
 - pt 287
 - \ptexfontname 1174
 - \ptexlineendmode 1175
 - \ptexminorversion 1176
 - \ptexrevision 1177
 - \ptextracingfonts 1178
 - \ptexversion 1179
 - \pxdimen 953
- Q**
- quark commands:
- \q_mark 152, 464, 17060, 34524, 34526, 34533, 34536, 34546
 - \q_nil .. 27, 28, 130, 152, 396, 846, 848, 850, 1545, 1548, 11224, 11226, 17060, 17114, 17133, 17139, 17154, 17155, 17161, 17185, 17189, 23569, 23570, 23572, 23574, 23576, 23578, 23584, 36366, 39914, 39920, 39921
 - \q_no_value 79, 96, 103–106, 151, 152, 160, 167, 198, 223, 340, 846, 848, 868, 869, 915, 972, 8960, 10411, 10424, 11133, 11286, 11389, 11392, 11395, 11398, 11427, 17060, 17122, 17143, 17149, 17777, 17785, 17797, 17823, 19342, 19357, 21066, 21193, 21199, 40704
 - \quark_if_nil:N 17112
 - \quark_if_nil:n 848, 849, 17130, 17150
 - \quark_if_nil:NTF 152, 387, 850, 17112, 36371
 - \quark_if_nil:nTF .. 152, 732, 847, 848, 850, 11228, 17130, 39928, 39941
 - \quark_if_nil_p:N 152, 17112
 - \quark_if_nil_p:n 152, 17130
 - \quark_if_no_value:N .. 17120, 17128
 - \quark_if_no_value:n 17140
 - \quark_if_no_value:NTF 152, 17112, 38433, 38437, 38507, 38511
 - \quark_if_no_value:nTF ... 152, 17130
 - \quark_if_no_value_p:N ... 152, 17112
 - \quark_if_no_value_p:n ... 152, 17130
 - \quark_if_recursion_tail_-break:NN ... 153, 849, 17100, 17100

- \quark_if_recursion_tail_-
 - break:nN ... [153](#), [849](#), [17100](#), [17106](#)
 - \quark_if_recursion_tail_stop:N .
 - [153](#), [386](#), [849](#), [1346](#), [17068](#), [17068](#),
[34870](#), [34903](#), [35751](#), [35804](#), [35830](#)
 - \quark_if_recursion_tail_stop:n .
 - [153](#),
[386](#), [848](#), [849](#), [5936](#), [6056](#), [17082](#),
[17082](#), [17098](#), [18188](#), [32147](#), [32444](#)
 - \quark_if_recursion_tail_stop_-
 - do:Nn
.. [153](#), [386](#), [849](#), [17068](#), [17074](#), [31432](#)
 - \quark_if_recursion_tail_stop_-
 - do:nn [153](#),
[386](#), [849](#), [17082](#), [17089](#), [17099](#),
[33408](#), [35290](#), [36113](#), [36147](#), [36182](#),
[36207](#), [36242](#), [36318](#), [36350](#), [39109](#)
 - \quark_new:N
.. [152](#), [386](#), [387](#), [852](#), [4442](#), [4443](#),
[8410](#), [8411](#), [10537](#), [11041](#), [11043](#),
[11044](#), [12461](#), [12462](#), [12463](#), [12464](#),
[12465](#), [13687](#), [13688](#), [14458](#), [17055](#),
[17055](#), [17060](#), [17061](#), [17062](#), [17063](#),
[17064](#), [17065](#), [17067](#), [18207](#), [18208](#),
[19866](#), [20743](#), [20744](#), [22491](#), [32664](#),
[32666](#), [32667](#), [41682](#), [41683](#), [42353](#)
- \q_recursion_stop [27](#),
[28](#), [153](#), [154](#), [396](#), [846](#), [1547](#), [1551](#),
[5932](#), [6051](#), [17064](#), [18177](#), [31427](#),
[32161](#), [32173](#), [32187](#), [32440](#), [33403](#),
[34890](#), [34925](#), [35286](#), [35295](#), [35302](#),
[35784](#), [35827](#), [36045](#), [36100](#), [39105](#)
- \q_recursion_tail
..... [153](#), [154](#), [846](#), [847](#),
[5932](#), [6050](#), [17064](#), [17070](#), [17076](#),
[17085](#), [17092](#), [17097](#), [17102](#), [17109](#),
[18177](#), [31427](#), [32160](#), [32172](#), [32186](#),
[32440](#), [33403](#), [34889](#), [34924](#), [35286](#),
[35783](#), [35826](#), [36044](#), [36100](#), [39104](#)
- \q_stop [27](#), [28](#), [41](#), [124](#), [151](#),
[152](#), [396](#), [846](#), [1546](#), [1549](#), [9204](#),
[9214](#), [11224](#), [11226](#), [13159](#), [17060](#),
[32189](#), [32190](#), [32191](#), [32192](#), [32193](#),
[32206](#), [32209](#), [32238](#), [32242](#), [32252](#),
[32277](#), [32307](#), [32458](#), [32492](#), [32504](#),
[32521](#), [32532](#), [32533](#), [32539](#), [32541](#),
[32542](#), [32544](#), [32547](#), [32561](#), [32568](#),
[36367](#), [36375](#), [36376](#), [36379](#), [39941](#)
- quark internal commands:
 - \q_bool_recursion_stop
..... [8410](#), [8413](#), [8526](#), [8552](#)
 - \q_bool_recursion_tail
..... [8410](#), [8526](#), [8552](#)
 - \q_char_no_value [19866](#)
 - \q_cs_nil [3073](#)
 - \q_cs_recursion_stop
..... [2755](#), [2759](#), [2770](#), [3066](#)
 - \q_debug_recursion_stop
..... [41682](#), [41685](#), [41932](#), [41937](#)
 - \q_debug_recursion_tail
..... [41682](#), [41932](#), [41937](#)
 - \q_file_nil
.. [11041](#), [11108](#), [11122](#), [11248](#), [11254](#)
 - \q_file_recursion_stop
..... [11043](#), [11087](#), [11098](#)
 - \q_file_recursion_tail
..... [11043](#), [11087](#), [11091](#)
 - \q_int_recursion_stop
.. [18207](#), [18971](#), [18988](#), [19031](#), [19058](#)
 - \q_int_recursion_tail
..... [18207](#), [18971](#), [18988](#), [19031](#)
 - \q_iow_nil [10537](#), [10788](#), [10795](#)
 - \q_keys_no_value
[1042](#), [22473](#), [22491](#), [23185](#), [23205](#),
[23208](#), [23222](#), [23227](#), [23242](#), [23247](#)
 - \q_prg_recursion_stop
..... [403](#), [1594](#), [1667](#), [1754](#)
 - \q_prg_recursion_tail
..... [403](#), [1667](#), [1677](#), [1754](#), [1773](#)
 - \q_prop_recursion_stop
..... [20743](#), [21484](#), [21572](#)
 - \q_prop_recursion_tail
..... [20743](#), [21483](#), [21571](#)
 - __quark_if_empty_if:n
.. [17130](#), [17132](#), [17142](#), [17152](#), [17292](#)
 - __quark_if_nil:w
..... [848](#), [17130](#), [17133](#), [17139](#)
 - __quark_if_no_value:w
..... [17130](#), [17143](#), [17149](#)
 - __quark_if_recursion_tail:w [847](#),
[852](#), [17082](#), [17085](#), [17092](#), [17096](#), [17109](#)
 - __quark_module_name:N
.... [853](#), [17158](#), [17181](#), [17310](#), [17312](#)
 - __quark_module_name:w
..... [17310](#), [17314](#), [17317](#)
 - __quark_module_name_end:w
..... [17310](#), [17325](#), [17328](#)
 - __quark_module_name_loop:w
..... [17310](#), [17318](#), [17319](#), [17323](#)
 - __quark_new_conditional:Nnnn ...
..... [17157](#), [17179](#), [17183](#), [17200](#)
 - __quark_new_conditional_N:Nnnn .
..... [17277](#), [17282](#)
 - __quark_new_conditional_n:Nnnn .
..... [17277](#), [17277](#)
 - __quark_new_conditional_N_-
aux:NNNn [17277](#), [17284](#), [17299](#)

- _quark_new_conditional_n-
 aux:NNNn [17277](#), [17279](#), [17287](#)
 - _quark_new_test:NNNn
 [17157](#), [17165](#), [17170](#), [17176](#)
 - _quark_new_test_aux:Nn
 [17158](#), [17159](#), [17169](#)
 - _quark_new_test_aux:nnNNnnnn . .
 [17157](#), [17172](#), [17193](#), [17201](#)
 - _quark_new_test_aux_do:nnNNnnnnNNn
 [851](#), [852](#), [17212](#), [17217](#), [17222](#),
 [17227](#), [17232](#), [17238](#), [17241](#), [17241](#)
 - _quark_new_test_define_break-
 ifx:nnNNNn [17239](#), [17254](#), [17275](#)
 - _quark_new_test_define_break-
 tl:nnNNNn [17223](#), [17254](#), [17273](#)
 - _quark_new_test_define-
 ifx:nnNNNn [851](#),
 [852](#), [17228](#), [17233](#), [17254](#), [17263](#), [17276](#)
 - _quark_new_test_define-
 tl:nnNNNn [851](#),
 [852](#), [17213](#), [17218](#), [17254](#), [17254](#), [17274](#)
 - _quark_new_test_N:Nnnn [17210](#), [17225](#)
 - _quark_new_test_n:Nnnn [17210](#), [17210](#)
 - _quark_new_test_NN:Nnnn
 [17210](#), [17236](#)
 - _quark_new_test_Nn:Nnnn
 [17210](#), [17230](#)
 - _quark_new_test_nN:Nnnn [17220](#)
 - _quark_new_test_nn:Nnnn
 [17210](#), [17215](#)
 - \q_quark_nil [17067](#)
 - _quark_quark_conditional-
 name:N [854](#), [17180](#), [17332](#), [17334](#)
 - _quark_quark_conditional-
 name:w [854](#), [17332](#), [17336](#), [17339](#)
 - _quark_test_define_aux:NNNNnnNNn
 [852](#), [17241](#), [17243](#), [17248](#)
 - _quark_tmp:w
 [854](#), [17310](#), [17331](#), [17332](#), [17342](#)
 - \q_regex_nil
 [4413](#), [4418](#), [4443](#), [4448](#), [5056](#),
 [5060](#), [5716](#), [5734](#), [5735](#), [5830](#), [5840](#)
 - \q_regex_recursion_stop
 [4442](#), [4445](#), [4447](#), [5716](#), [5735](#), [7721](#)
 - \q_str_nil
 [811](#), [14458](#), [15548](#), [15555](#), [15570](#), [15597](#)
 - \q_str_recursion_stop
 [13687](#), [14294](#), [14302](#), [14307](#)
 - \q_str_recursion_tail
 [765](#), [13687](#), [13934](#),
 [13943](#), [13960](#), [13980](#), [14002](#), [14294](#)
 - \q_text_nil [32664](#), [33238](#), [33239](#)
 - \q_text_recursion_stop
 [32666](#), [32669](#), [33048](#),
 [33145](#), [33169](#), [33189](#), [33441](#), [33444](#),
 [33453](#), [33487](#), [33490](#), [33513](#), [33529](#),
 [33538](#), [33596](#), [33659](#), [33668](#), [33760](#),
 [33768](#), [34001](#), [34009](#), [34028](#), [34036](#),
 [34127](#), [34135](#), [34216](#), [34221](#), [34464](#),
 [34469](#), [34496](#), [34505](#), [34558](#), [34564](#),
 [34682](#), [34687](#), [34739](#), [34744](#), [34781](#),
 [34786](#), [34817](#), [34826](#), [34940](#), [34944](#),
 [34953](#), [34982](#), [34998](#), [35007](#), [35069](#),
 [35077](#), [35109](#), [35114](#), [35251](#), [35259](#),
 [35303](#), [35520](#), [35533](#), [35542](#), [35560](#),
 [35573](#), [35575](#), [35592](#), [35601](#), [35629](#)
 - \q_text_recursion_tail [32666](#),
 [32816](#), [33047](#), [33145](#), [33169](#), [33189](#),
 [33441](#), [33487](#), [33490](#), [33512](#), [33596](#),
 [34940](#), [34981](#), [35520](#), [35559](#), [35629](#)
 - \q_tl_mark
 [726](#), [12461](#), [12652](#), [12654](#), [12656](#), [12658](#)
 - \q_tl_nil [726](#), [12461](#), [12690](#)
 - \q_tl_recursion_stop [12464](#)
 - \q_tl_recursion_tail [12464](#), [13387](#)
 - \q_tl_stop [726](#), [12461](#), [12689](#)
 - \quitvmode [678](#)
- R**
- \r [33333](#), [35799](#),
 [35818](#), [35842](#), [35868](#), [35992](#), [35993](#)
 - \radical [380](#)
 - \raise [381](#)
 - \rand [286](#)
 - \randint [286](#)
 - \randomseed [954](#)
 - \read [382](#)
 - \readline [518](#)
 - \readpapersizespecial [1180](#)
 - \ref [32936](#), [32946](#)
 - regex commands:
 \regex_const:Nn [57](#), [7321](#), [7331](#)
 \regex_count:NnN
 [58](#), [7371](#), [7373](#), [7376](#), [18403](#), [18409](#)
 \regex_count:nnN [58](#), [174](#),
 [579](#), [7371](#), [7371](#), [7375](#), [18391](#), [18397](#)
 \regex_extract_all:NnN
 [59](#), [7391](#), [7407](#), [17506](#)
 \regex_extract_all:nnN
 [50](#), [59](#), [158](#), [490](#), [7391](#), [7407](#), [17503](#)
 \regex_extract_all:NnNTF [59](#), [7391](#)
 \regex_extract_all:nnNTF [59](#), [7391](#)
 \regex_extract_once:NnN
 [59](#), [7391](#), [7405](#), [17500](#)
 \regex_extract_once:nnN
 [59](#), [158](#), [7391](#), [7405](#), [17497](#)
 \regex_extract_once:NnNTF [59](#), [7391](#)
 \regex_extract_once:nnNTF [53](#), [59](#), [7391](#)

- \regex_gset:Nn 57, [7321](#), [7326](#)
 - \regex_if_match:Nn [7365](#), [7370](#)
 - \regex_if_match:nn [7359](#), [7364](#)
 - \regex_if_match:NnTF
 - 58, [7359](#), [12923](#), [41501](#),
 - [41502](#), [41503](#), [41504](#), [41505](#), [41506](#)
 - \regex_if_match:nnTF 58,
 - [117](#), [581](#), [590](#), [7359](#), [12916](#), [41495](#),
 - [41496](#), [41497](#), [41498](#), [41499](#), [41500](#)
 - \regex_log:N 57, [533](#), [7336](#), [7347](#)
 - \regex_log:n 57, [7336](#), [7337](#)
 - \regex_match:Nn [41509](#)
 - \regex_match:nn [41507](#)
 - \regex_match:NnTF
 - [41495](#), [41502](#), [41504](#), [41506](#)
 - \regex_match:nnTF
 - [41495](#), [41496](#), [41498](#), [41500](#)
 - \regex_match_case:nn
 - . . . 58, [61](#), [512](#), [542](#), [7377](#), [7385](#), [7515](#)
 - \regex_match_case:nnTF . . 58, [7377](#),
 - [7377](#), [7386](#), [7387](#), [7388](#), [7389](#), [7390](#)
 - \regex_new:N 57,
 - [493](#), [7315](#), [7315](#), [7317](#), [7318](#), [7319](#), [7320](#)
 - \regex_replace_all:NnN
 - 60, [7391](#), [7411](#), [12741](#)
 - \regex_replace_all:nnN 50,
 - [60](#), [128](#), [202](#), [578](#), [7391](#), [7411](#), [12738](#)
 - \regex_replace_all:NnNTF . . . 60, [7391](#)
 - \regex_replace_all:nnNTF . . . 60, [7391](#)
 - \regex_replace_case_all:nN
 - 61, [7436](#), [7441](#), [7453](#)
 - \regex_replace_case_all:nNTF . . .
 - 61, [7436](#),
 - [7436](#), [7454](#), [7455](#), [7456](#), [7457](#), [7458](#)
 - \regex_replace_case_once:nN
 - 61, [7413](#), [7418](#), [7430](#)
 - \regex_replace_case_once:nNTF . . .
 - 61, [7413](#),
 - [7413](#), [7431](#), [7432](#), [7433](#), [7434](#), [7435](#)
 - \regex_replace_once:NnN
 - 60, [7391](#), [7409](#), [12735](#)
 - \regex_replace_once:nnN 59–
 - [61](#), [128](#), [216](#), [577](#), [7391](#), [7409](#), [12732](#)
 - \regex_replace_once:NnNTF . . 60, [7391](#)
 - \regex_replace_once:nnNTF
 - 60, [594](#), [7391](#)
 - \regex_set:Nn . . 49, 57, 58, [7321](#), [7321](#)
 - \regex_show:N . . 57, [521](#), [533](#), [7336](#), [7346](#)
 - \regex_show:n . . 50, 55, 57, [7336](#), [7336](#)
 - \regex_split:NnN . . 60, [7391](#), [7412](#), [17512](#)
 - \regex_split:nnN
 - 60, [159](#), [7391](#), [7412](#), [17509](#)
 - \regex_split:NnNTF 60, [7391](#)
 - \regex_split:nnNTF 60, [7391](#)
 - \g_tmpa_regex 62, [7317](#)
 - \l_tmpa_regex 62, [7317](#)
 - \g_tmpb_regex 62, [7317](#)
 - \l_tmpb_regex 62, [7317](#)
- regex internal commands:
- __regex_A_test: . . 505, [5324](#), [5346](#),
 - [5962](#), [5965](#), [5971](#), [6089](#), [6570](#), [6603](#)
 - __regex_action_cost:n 541,
 - [545](#), [6359](#), [6360](#), [6368](#), [6817](#), [6843](#), [6843](#)
 - __regex_action_free:n . . 541, [553](#),
 - [6382](#), [6388](#), [6389](#), [6400](#), [6458](#), [6462](#),
 - [6487](#), [6512](#), [6516](#), [6519](#), [6547](#), [6555](#),
 - [6565](#), [6579](#), [6622](#), [6815](#), [6819](#), [6819](#)
 - __regex_action_free_aux:nn
 - [6819](#), [6820](#), [6822](#), [6823](#)
 - __regex_action_free_group:n . . .
 - [541](#), [553](#), [6408](#), [6527](#), [6530](#), [6819](#), [6821](#)
 - __regex_action_start_wildcard:N
 - 541, [6243](#), [6263](#), [6812](#), [6812](#)
 - __regex_action_submatch:nN
 - 541, [6267](#), [6289](#),
 - [6481](#), [6482](#), [6620](#), [6868](#), [6870](#), [6870](#)
 - __regex_action_submatch_aux:w . . .
 - [6870](#), [6872](#), [6875](#)
 - __regex_action_submatch_auxii:w
 - [6870](#), [6881](#), [6886](#)
 - __regex_action_submatch_–
 - auxiii:w [6870](#), [6882](#), [6887](#), [6888](#), [6889](#)
 - __regex_action_submatch_auxiv:w
 - [6870](#)
 - __regex_action_success:
 - . . . 541, [6246](#), [6292](#), [6310](#), [6891](#), [6891](#)
 - __regex_action_wildcard: 558
 - \l__regex_added_begin_int
 - [7470](#), [7609](#), [7617](#), [7621](#),
 - [7675](#), [7804](#), [7809](#), [7813](#), [7824](#), [7839](#)
 - \l__regex_added_end_int
 - [7470](#), [7611](#), [7617](#), [7622](#),
 - [7676](#), [7806](#), [7809](#), [7814](#), [7826](#), [7840](#)
 - \c__regex_all_catcodes_int
 - [4870](#), [4982](#), [5086](#), [5682](#)
 - \c__regex_ascii_lower_int
 - [4441](#), [4502](#), [4507](#)
 - \c__regex_ascii_max_control_int . .
 - [4438](#), [4618](#)
 - \c__regex_ascii_max_int
 - [4438](#), [4611](#), [4619](#), [4810](#)
 - \c__regex_ascii_min_int
 - [4438](#), [4610](#), [4617](#)
 - __regex_assertion:Nn . . 505, [519](#),
 - [551](#), [5320](#), [5342](#), [5951](#), [6082](#), [6570](#), [6570](#)
 - __regex_b_test: 505,
 - [551](#), [5332](#), [5334](#), [5968](#), [6087](#), [6570](#), [6588](#)

```

\l_regex_balance_int .....
    ..... 493, 565, 588, 4437,
    6967, 6999, 7258, 7275, 7482, 7495,
    7497, 7498, 7755, 7781, 7805, 7807
\g_regex_balance_intarray .....
    ..... 490, 579, 6946, 6953, 7469, 7494
\g_regex_balance_tl .. 565, 6909,
    6968, 6998, 7024, 7041, 7051, 7126
\l_regex_begin_flag .....
    ..... 7460, 7600, 7610, 7653
\_regex_branch:n ..... 505, 523,
    547, 4434, 4987, 5062, 5492, 5545,
    5730, 5840, 5848, 5932, 5934, 5937,
    6064, 6453, 6453, 42455, 42456, 42457
\_regex_break_point:TF .....
    ..... 494, 518, 545, 4450, 4451,
    4452, 4456, 6359, 6360, 6576, 6593
\_regex_break_true:w .....
    . 494, 495, 4450, 4450, 4456, 4461,
    4468, 4475, 4479, 4486, 4492, 4539,
    4551, 4567, 5295, 6600, 6606, 6612
\_regex_build:N .....
    ..... 577, 6226, 6228, 7367,
    7374, 7394, 7398, 42424, 42427, 42429
\_regex_build:n ..... 542,
    577, 6226, 6226, 7361, 7372, 7393, 7396
\_regex_build_aux:NN . 589, 6226,
    6229, 6233, 6235, 7861, 7880, 7950
\_regex_build_aux:Nn .....
    589, 6226, 6227, 6230, 7852, 7870, 7942
\_regex_build_for_cs:n .....
    4562, 6299, 6299, 42431, 42434, 42436
\_regex_build_new_state: .....
    ..... 6240, 6241, 6260, 6261,
    6265, 6302, 6303, 6332, 6332, 6341,
    6373, 6407, 6411, 6455, 6470, 6475,
    6514, 6533, 6568, 6572, 6617, 42449
\l_regex_build_tl .... 523, 594,
    4431, 4979, 4986, 5004, 5009, 5012,
    5013, 5016, 5017, 5020, 5080, 5083,
    5123, 5137, 5141, 5264, 5278, 5319,
    5341, 5354, 5386, 5399, 5403, 5485,
    5488, 5491, 5497, 5498, 5501, 5544,
    5834, 5838, 5845, 5851, 5872, 5888,
    5906, 6063, 6120, 6123, 6134, 6164,
    6179, 6183, 6186, 6192, 6966, 6989,
    7000, 7003, 7054, 7123, 7180, 7183,
    7197, 7265, 8008, 8011, 8019, 8022
\_regex_build_transition-
    left:NNN 6328, 6328, 6516, 6530, 6547
\_regex_build_transition-
    right:nNn ..... 6328,
    6330, 6374, 6408, 6458, 6462,
    6487, 6512, 6519, 6527, 6555, 6565
\_regex_build_transitions-
    laziness:NNNNN .....
    ..... 6339, 6339, 6381, 6387, 6399
\l_regex_capturing_group_int ...
    ..... 490, 540, 587,
    6225, 6238, 6276, 6281, 6284, 6424,
    6426, 6437, 6438, 6446, 6447, 6450,
    6714, 6787, 6788, 6861, 6880, 7114,
    7118, 7706, 7727, 7735, 7786, 7794
\g_regex_case_balance_tl .....
    ..... 7029, 7032, 7038, 7042, 7050
\_regex_case_build:n .....
    581, 6250, 6250, 6255, 7424, 7447, 7521
\_regex_case_build_aux:Nn .....
    ..... 6250, 6252, 6256
\_regex_case_build_loop:n .....
    ..... 6250, 6274, 6279
\l_regex_case_changed_char_int .
    ..... 495, 4478,
    4490, 4491, 4498, 4502, 4507, 6635
\g_regex_case_int .....
    ..... 577, 578, 6248, 6253, 6270,
    6273, 6290, 6291, 7381, 7425, 7717
\l_regex_case_max_group_int ...
    ..... 6249, 6269, 6276, 6283, 6284
\_regex_case_replacement:n ....
    ..... 7028, 7030, 7046, 7448
\_regex_case_replacement_aux:n .
    ..... 7040, 7047
\g_regex_case_replacement_tl ...
    ..... 7028, 7038, 7044, 7049
\c_regex_catcode_A_int ..... 4870
\c_regex_catcode_B_int ..... 4870
\c_regex_catcode_C_int ..... 4870
\c_regex_catcode_D_int ..... 4870
\c_regex_catcode_E_int ..... 4870
\c_regex_catcode_in_class_mode-
    int 4860, 4971, 5353, 5514, 5607, 5636
\c_regex_catcode_L_int ..... 4870
\c_regex_catcode_M_int ..... 4870
\c_regex_catcode_mode_int .....
    .. 4860, 4967, 5040, 5385, 5605, 5634
\c_regex_catcode_O_int ..... 4870
\c_regex_catcode_P_int ..... 4870
\c_regex_catcode_S_int ..... 4870
\c_regex_catcode_T_int ..... 4870
\c_regex_catcode_U_int ..... 4870
\l_regex_catcodes_bool .....
    ..... 4867, 5641, 5645, 5680
\l_regex_catcodes_int .....
    ..... 506, 4867, 4983, 5085,
    5087, 5093, 5372, 5389, 5489, 5502,
    5601, 5638, 5673, 5675, 5681, 5682
\_regex_char_if_alphanumeric:N 4833

```


__regex_char_if_alphanumeric:NTF	__regex_compile:w
..... 4804, 5033, 7232	 511, 4976, 4976, 5024, 5687
__regex_char_if_special:N ... 4804	__regex_compile\$: 5315
__regex_char_if_special:NTF ...	__regex_compile(: 5509
..... 4804, 5029	__regex_compile): 5548
__regex_chk_c_allowed:TF	__regex_compile.: 5286
..... 4953, 4953, 5594	__regex_compile_/A: 5315
__regex_class:NnnnN	__regex_compile_/B: 5315
..... 505, 513, 514, 520,	__regex_compile_/b: 5315
4435, 5081, 5380, 5381, 5387, 5747,	__regex_compile_/c: 5593
5880, 5890, 5952, 6079, 6353, 6353	__regex_compile_/D: 5298
\c_regex_class_mode_int	__regex_compile_/d: 5298
..... 4860, 4957, 4972	__regex_compile_/G: 5315
__regex_class_repeat:n	__regex_compile_/H: 5298
... 546, 6363, 6369, 6369, 6385, 6394	__regex_compile_/h: 5298
__regex_class_repeat:nN	__regex_compile_/K: 5903
..... 6364, 6378, 6378	__regex_compile_/N: 5298
__regex_class_repeat:nnN	__regex_compile_/S: 5298
..... 6365, 6392, 6392	__regex_compile_/s: 5298
__regex_clean_assertion:Nn	__regex_compile_/u: 5767
..... 5909, 5951, 5959	__regex_compile_/V: 5298
__regex_clean_bool:n	__regex_compile_/v: 5298
.. 5909, 5909, 5961, 5976, 5980, 5988	__regex_compile_/W: 5298
__regex_clean_branch:n	__regex_compile_/w: 5298
..... 5909, 5937, 5940	__regex_compile_/Z: 5315
__regex_clean_branch_loop:n	__regex_compile_/z: 5315
5909, 5942, 5945, 5950, 5972, 5981, 5989	__regex_compile[: 5364
__regex_clean_class:n	__regex_compile]: 5348
..... 5909, 5977, 5991, 6002, 6023	__regex_compile^: 5315
__regex_clean_class:NnnnN	__regex_compile_abort_tokens:n
..... 5909, 5952, 5974	.. 5096, 5096, 5104, 5130, 5469, 5479
__regex_clean_class_loop:nnn	__regex_compile_anchor_letter:NNN
..... 5909,	 5315, 5315,
5992, 5993, 6004, 6014, 6024, 6038	5324, 5326, 5328, 5330, 5332, 5334
__regex_clean_exact_cs:n	__regex_compile_c[:w
..... 5909, 5999, 6045	 5630
__regex_clean_exact_cs:w	__regex_compile_c_C:NN
..... 5909, 6049, 6054, 6058	 5609, 5618, 5618
__regex_clean_group:nnnN	__regex_compile_c_lbrack_add:N
..... 5909, 5953, 5954, 5955, 5983	 5630, 5656, 5671
__regex_clean_int:n	__regex_compile_c_lbrack_end:
... 5909, 5915, 5918, 5978, 5979,	 5630, 5663, 5667, 5678
5986, 5987, 6000, 6001, 6013, 6023	__regex_compile_c_lbrack-
__regex_clean_int_aux:N	loop:NN 5630, 5642, 5646, 5650, 5658
..... 5909, 5919, 5921	__regex_compile_c_test:NN
__regex_clean_regex:n	 5593, 5594, 5595
..... 5909, 5929, 5985, 5998, 7351	__regex_compile_class:NN
__regex_clean_regex_loop:w	 5394, 5400, 5404, 5407
..... 5909, 5931, 5934, 5938	__regex_compile_class:TFNN
__regex_command_K:	 520, 5379, 5390, 5394, 5394
... 505, 5906, 5950, 6080, 6615, 6615	__regex_compile_class_catcode:w
__regex_compile:n	 5371, 5383, 5383
5022, 5022,	__regex_compile_class_normal:w
5058, 6232, 7323, 7328, 7333, 7340	 5374, 5377, 5377


```

__regex_compile_class_posix:NNNNw
    ..... 5413, 5419, 5432
__regex_compile_class_posix_-
    end:w ..... 5413, 5450, 5452
__regex_compile_class_posix_-
    loop:w . 5413, 5438, 5443, 5446, 5449
__regex_compile_class_posix_-
    test:w ..... 5367, 5413, 5413
__regex_compile_cs_aux:Nn .....
    ..... 5702, 5715, 5728, 5736
__regex_compile_cs_aux:NNnnN ..
    ..... 5702, 5733, 5743, 5756
__regex_compile_end: .....
    ..... 511, 4976, 4989, 5049, 5711
__regex_compile_end_cs: .....
    ..... 5045, 5702, 5706, 5709
__regex_compile_escaped:N .....
    ..... 5034, 5065, 5070
__regex_compile_group_begin:N ..
    .. 5483, 5483, 5531, 5536, 5554, 5556
__regex_compile_group_end: ....
    ..... 5483, 5494, 5551
__regex_compile_if_quantifier:TFw
    ..... 5105, 5105, 5831, 5843
__regex_compile_lparen:w 5518, 5522
__regex_compile_one:n .....
    ..... 5075, 5075, 5232, 5238,
    ..... 5290, 5301, 5304, 5314, 5460, 5718
__regex_compile_quantifier:w ...
    ..... 5094,
    ..... 5112, 5112, 5359, 5503, 5836, 5852
__regex_compile_quantifier*:w 5146
__regex_compile_quantifier+:w 5146
__regex_compile_quantifier?:w 5146
__regex_compile_quantifier_-
    abort:nNN .....
    .. 5121, 5126, 5156, 5175, 5188, 5211
__regex_compile_quantifier_-
    braced_auxi:w ... 5152, 5155, 5158
__regex_compile_quantifier_-
    braced_auxii:w .. 5152, 5171, 5180
__regex_compile_quantifier_-
    braced_auxiii:w . 5152, 5170, 5193
__regex_compile_quantifier_-
    laziness:nnNN . 515, 5133, 5133,
    ..... 5147, 5149, 5151, 5164, 5184, 5206
__regex_compile_quantifier_-
    none: .. 5117, 5119, 5121, 5121, 5128
__regex_compile_range:Nw .....
    ..... 5230, 5243, 5257
__regex_compile_raw:N 4909, 5030,
    ..... 5034, 5036, 5068, 5073, 5101, 5223,
    ..... 5225, 5225, 5245, 5289, 5339, 5362,
    ..... 5410, 5430, 5448, 5506, 5511, 5516,
    ..... 5532, 5542, 5550, 5568, 5569, 5570,
    ..... 5576, 5587, 5588, 5589, 5597, 5652,
    ..... 5700, 5707, 5772, 5788, 5789, 5795
__regex_compile_raw_error:N ...
    ..... 5220, 5220, 5317, 5770, 5907
__regex_compile_special:N . 507,
    ..... 5030, 5065, 5065, 5107, 5114, 5135,
    ..... 5162, 5167, 5182, 5195, 5229, 5247,
    ..... 5397, 5415, 5434, 5454, 5455, 5524,
    ..... 5559, 5577, 5620, 5639, 5779, 5798
__regex_compile_special_group_-
    -:w ..... 5557
__regex_compile_special_group_-
    ::w ..... 5553
__regex_compile_special_group_-
    i:w ..... 5557, 5557
__regex_compile_special_group_-
    l:w ..... 5553
__regex_compile_u_brace:NNN ...
    ..... 5773, 5774, 5777, 5777
__regex_compile_u_end: .....
    ..... 5774, 5841, 5841
__regex_compile_u_in_cs: .....
    ..... 5862, 5865, 5865
__regex_compile_u_in_cs_aux:n ..
    ..... 5875, 5878
__regex_compile_u_loop:NN .....
    ..... 5783, 5793, 5793, 5796, 5808
__regex_compile_u_not_cs: .....
    ..... 5860, 5884, 5884
__regex_compile_u_payload: ....
    ..... 531, 5841, 5850, 5854, 5856
__regex_compile_ur:n .....
    ..... 531, 5819, 5826, 5828
__regex_compile_ur_aux:w .....
    ..... 5819, 5830, 5840
__regex_compile_ur_end: .....
    ..... 5773, 5787, 5819, 5819
__regex_compile_use:n .....
    ..... 5051, 5051, 6282
__regex_compile_use_aux:w 5055, 5060
__regex_compile_|: ..... 5540
__regex_compute_case_changed_-
    char: ..... 4496, 4496, 4512, 6758
__regex_count:nnN .....
    ..... 7372, 7374, 7527, 7527
\l_regex_cs_flag ..... 5702
\c_regex_cs_in_class_mode_int ..
    ..... 4860, 5693
\c_regex_cs_mode_int ... 4860, 5691
\l_regex_curr_analysis_tl .....
    ..... 555, 6649, 6695, 6722, 6729, 6763, 6764
\l_regex_curr_catcode_int .....
    .. 4518, 4537, 4545, 4557, 6635, 6761

```

\l_regex_curr_char_int
 557, 4460, 4466, 4467,
 4474, 4484, 4485, 4498, 4499, 4500,
 4501, 4506, 4538, 5294, 6309, 6591,
 6599, 6635, 6718, 6757, 6760, 6776
 _regex_curr_cs_to_str:
 4394, 4394, 4548, 4565
 \l_regex_curr_pos_int
 492, 557, 6611, 6630, 6706, 6717,
 6756, 6890, 6898, 7483, 7488, 7492,
 7493, 7495, 7993, 7998, 8002, 8003
 \l_regex_curr_state_int 554, 560,
 6641, 6794, 6795, 6797, 6802, 6805,
 6827, 6832, 6837, 6838, 6846, 42479
 \l_regex_curr_submatches_tl ...
 6642, 6713, 6807,
 6839, 6840, 6851, 6873, 6877, 6902
 \l_regex_curr_token_tl
 4397, 6635, 6759
 \l_regex_default_catcodes_int ..
 506, 4867,
 4981, 4983, 5093, 5389, 5489, 5502
 _regex_disable_submatches: 4561,
 5688, 6865, 6865, 7504, 7530, 7891
 \l_regex_empty_success_bool ...
 6652, 6698, 6702, 6896, 7591
 \l_regex_end_flag
 7460, 7601, 7612, 7661
 _regex_escape_\w 4684
 _regex_escape_\wscan_stop::w 4684
 _regex_escape_\a:w 4684
 _regex_escape_\e:w 4684
 _regex_escape_\f:w 4684
 _regex_escape_\n:w 4684
 _regex_escape_\r:w 4684
 _regex_escape_\t:w 4684
 _regex_escape_\x:w 4703
 _regex_escape_\w 4668
 _regex_escape_\wscan_stop::w . 4684
 _regex_escape_escaped:N
 4654, 4678, 4681, 4682
 _regex_escape_loop:N 500,
 4661, 4668, 4668, 4672, 4675, 4679,
 4703, 4742, 4753, 4754, 4774, 4783
 _regex_escape_raw:N
 501, 4655, 4681, 4683, 4692,
 4694, 4696, 4698, 4700, 4702, 4716
 _regex_escape_unescaped:N
 4653, 4671, 4681, 4681
 _regex_escape_use:nnn 42422
 _regex_escape_use:nnnn 499, 511,
 4649, 4649, 5027, 6969, 42415, 42418
 _regex_escape_x:N
 501, 4741, 4745, 4745
 _regex_escape_x_end:w
 501, 4703, 4705, 4708
 _regex_escape_x_large:n 4703
 _regex_escape_x_loop:N
 501, 4738, 4757, 4757, 4766, 4769
 _regex_escape_x_loop_error: . 4757
 _regex_escape_x_loop_error:n ..
 4763, 4775, 4780
 _regex_escape_x_test:N
 501, 4706, 4720, 4720, 4728
 _regex_escape_x_testii:N
 4720, 4730, 4735
 \l_regex_every_match_tl
 6651, 6733, 6743, 6780
 _regex_extract:
 581, 593, 7545, 7552,
 7565, 7702, 7702, 7752, 7776, 7966
 _regex_extract_all:nnN
 7406, 7539, 7549
 _regex_extract_aux:w
 7702, 7719, 7724, 7740
 _regex_extract_check:n
 7666, 7668, 7671
 _regex_extract_check:w
 583, 584, 7613, 7666, 7666, 7677
 _regex_extract_check_end:w ...
 585, 7666, 7682, 7694
 _regex_extract_check_loop:w ...
 7666, 7680, 7687, 7692, 7695
 _regex_extract_once:nnN
 7404, 7539, 7539
 _regex_extract_seq:N
 7598, 7625, 7627
 _regex_extract_seq:NNn
 7598, 7631, 7635
 _regex_extract_seq_aux:n
 7606, 7642, 7642
 _regex_extract_seq_aux:ww ...
 7642, 7645, 7648
 _regex_extract_seq_loop:Nw ...
 7598, 7630, 7637, 7640
 \l_regex_fresh_thread_bool
 555, 560, 6621,
 6627, 6652, 6774, 6814, 6816, 6897
 _regex_G_test:
 505, 5326, 5966, 6090, 6570, 6609
 _regex_get_digits:NtFw
 4895, 4895, 5154, 5169
 _regex_get_digits_loop:nw ...
 4898, 4901, 4904
 _regex_get_digits_loop:w ... 4895
 _regex_group:nnnN
 505, 523, 5531, 5536,
 5822, 5953, 6073, 6244, 6421, 6421

```

\\__regex_group_aux:nnnnN .....
    ..... 547, 6404, 6404,
    6423, 6431, 6434, 42451, 42452, 42453
\\__regex_group_aux:nnnnN ..... 547
\\__regex_group_end_extract_seq:N
    ... 584, 7547, 7556, 7596, 7598, 7598
\\__regex_group_end_replace:N ...
    ..... 7767, 7800, 7802, 7802
\\__regex_group_end_replace_-
    check:n ..... 588, 7802, 7832, 7835
\\__regex_group_end_replace_-
    check:w ..... 588, 7802, 7821, 7830
\\__regex_group_end_replace_try: .
    ..... 588, 7802, 7808, 7819, 7841
\\l_regex_group_level_int . 4859,
    4980, 4998, 5000, 5002, 5490, 5496
\\__regex_group_no_capture:nnnN ..
    ..... 505, 5554, 5822, 5823,
    5835, 5847, 5954, 6075, 6421, 6430
\\__regex_group_repeat:nn .....
    ..... 6416, 6465, 6465
\\__regex_group_repeat:nnN .....
    ..... 6417, 6505, 6505
\\__regex_group_repeat:nnnN .....
    ..... 6418, 6536, 6536
\\__regex_group_repeat_aux:n ....
    548, 550, 6472, 6485, 6485, 6523, 6540
\\__regex_group_resetting:nnnN ...
    505, 5556, 5823, 5955, 6077, 6432, 6432
\\__regex_group_resetting_-
    loop:nnNn .. 6432, 6436, 6444, 6449
\\__regex_group_submatches:nnN ...
    .. 6473, 6478, 6478, 6508, 6524, 6538
\\__regex_hexadecimal_use:NTF ...
    ..... 4740, 4752, 4765, 4785, 4785
\\__regex_if_end_range:NNTF .....
    ..... 5243, 5243, 5259
\\__regex_if_in_class: ..... 4916
\\__regex_if_in_class:TF ... 4916,
    4991, 5078, 5094, 5227, 5288, 5350,
    5366, 5511, 5542, 5550, 8086, 8099
\\__regex_if_in_class_or_catcode:TF
    ..... 4934, 4934, 5317, 5339, 5769
\\__regex_if_in_cs:TF .....
    .. 4924, 4924, 5698, 5705, 8084, 8093
\\__regex_if_match:nn .....
    ..... 7361, 7367, 7501, 7501, 7520
\\__regex_if_raw_digit:NNTF .....
    ..... 4897, 4903, 4907, 4907
\\__regex_if_two_empty_matches:TF
    ... 555, 6652, 6654, 6703, 6709, 6893
\\__regex_if_within_catcode: .. 4945
\\__regex_if_within_catcode:TF ...
    ..... 4945, 5369
\\__regex_input_item:n .....
    ..... 589, 593, 594, 7847,
    7848, 7908, 7930, 7971, 7994, 8003
\\l_regex_input_tl .....
    ..... 590, 592, 593, 7847,
    7903, 7907, 7929, 7931, 7992, 7996
\\__regex_int_eval:w .....
    ..... 4357, 4357, 4400, 4527,
    4791, 5673, 6329, 6331, 6345, 6346,
    6348, 6349, 6491, 6581, 6624, 6798,
    6846, 6859, 6923, 6924, 6935, 6945,
    7124, 7127, 7729, 7733, 8020, 8025
\\__regex_intarray_item:NnTF ....
    ..... 4399, 4399, 6946, 6953
\\__regex_intarray_item_aux:nNTF .
    ..... 4399, 4400, 4401
\\__regex_item_caseful_equal:n ...
    ..... 505, 4458,
    4458, 4578, 4579, 4583, 4584, 4585,
    4586, 4587, 4596, 4601, 4619, 4637,
    4984, 5581, 5749, 5881, 6000, 6091
\\__regex_item_caseful_range:nn ..
    ..... 505,
    4458, 4464, 4575, 4590, 4593, 4594,
    4595, 4609, 4616, 4623, 4625, 4627,
    4630, 4631, 4632, 4633, 4638, 4641,
    4646, 4647, 4985, 5583, 6008, 6093
\\__regex_item_caseless_equal:n ..
    ... 505, 4472, 4472, 5562, 6001, 6098
\\__regex_item_caseless_range:nn .
    ... 505, 4472, 4482, 5564, 6009, 6100
\\__regex_item_catcode: .....
    ..... 4515, 4515, 4527
\\__regex_item_catcode:n ..... 4525
\\__regex_item_catcode:nTF .. 505,
    520, 4515, 4534, 5087, 5391, 6019, 6105
\\__regex_item_catcode_reverse:nTF
    ... 505, 4515, 4533, 5392, 6020, 6107
\\__regex_item_cs:n .....
    ... 505, 4555, 4555, 5721, 5998, 6114
\\__regex_item_equal:n .....
    ..... 4513, 4513, 4984, 5233,
    5239, 5267, 5280, 5281, 5561, 5580
\\__regex_item_exact:nn .....
    505, 532, 4535, 4535, 5896, 6010, 6111
\\__regex_item_exact_cs:n ... 505,
    528, 4535, 4543, 5723, 5893, 5999, 6113
\\__regex_item_range:nn .....
    .. 4513, 4514, 4985, 5269, 5563, 5582
\\__regex_item_reverse:n .....
    ..... 505, 521, 4453, 4453, 4534,
    4600, 5305, 5462, 6002, 6109, 6594
\\l_regex_last_char_int .....
    ..... 6591, 6605, 6635, 6757, 6899

```

- \l_regex_last_char_success_int .
..... 6635, 6693, 6718, 6899
- \l_regex_left_state_int
..... 6221, 6242,
6262, 6266, 6317, 6324, 6335, 6342,
6345, 6346, 6348, 6349, 6375, 6383,
6386, 6409, 6457, 6459, 6469, 6489,
6509, 6511, 6539, 6542, 6545, 6548,
6560, 6573, 6582, 6618, 6625, 42442
- \l_regex_left_state_seq
..... 6221, 6316, 6323, 6456
- _regex_maplike_break:
..... 492, 590, 4408, 4408, 4409,
6665, 6679, 6724, 6738, 6746, 7911
- _regex_match:n
6658, 6658, 7507, 7534, 7544, 7554,
7580, 7749, 7778, 42460, 42463, 42464
- _regex_match_case:nnTF
..... 7379, 7510, 7510
- _regex_match_case_aux:nn 7510, 7526
- \l_regex_match_count_int
..... 579, 581, 7459, 7531, 7532, 7537
- _regex_match_cs:n
4565, 6658, 6667, 42467, 42470, 42471
- _regex_match_init:
.. 6658, 6660, 6670, 6681, 7902, 42475
- _regex_match_once_init:
.. 6661, 6671, 6700, 6700, 6750, 7904
- _regex_match_once_init_aux: ...
..... 6720, 6726
- _regex_match_one_active:n
..... 6753, 6771, 6782
- _regex_match_one_token:nnN ...
.. 557, 560, 590, 6663, 6664, 6675,
6676, 6678, 6723, 6753, 6753, 7909
- \l_regex_match_success_bool ...
... 555, 6655, 6712, 6737, 6745, 6895
- \l_regex_matched_analysis_tl ...
555, 6649, 6694, 6719, 6728, 6762, 6900
- \l_regex_max_pos_int
..... 564, 6630, 7488,
7586, 7592, 7765, 7798, 7984, 7998
- \l_regex_max_state_int 540, 543,
602, 6218, 6239, 6259, 6294, 6296,
6297, 6301, 6334, 6336, 6337, 6396,
6468, 6488, 6490, 6498, 6542, 6548,
6556, 6566, 6685, 8358, 42444, 42445
- \l_regex_max_thread_int
..... 6645, 6669,
6715, 6767, 6770, 6775, 6852, 6860
- _regex_maybe_compute_ccc:
..... 4477, 4489, 4510, 4512, 6758
- \l_regex_min_pos_int
..... 564, 6630, 6691, 6692
- \l_regex_min_state_int 543, 6218,
6239, 6259, 6301, 6685, 6716, 8357
- \l_regex_min_submatch_int
..... 579, 583,
587, 6696, 6697, 7462, 7605, 7785, 7793
- \l_regex_min_thread_int
.. 6645, 6669, 6715, 6767, 6769, 6775
- \l_regex_mode_int 4860,
4918, 4926, 4928, 4936, 4938, 4947,
4955, 4957, 4967, 4968, 4970, 4972,
5026, 5040, 5042, 5352, 5356, 5357,
5358, 5385, 5396, 5513, 5603, 5604,
5632, 5633, 5689, 5690, 5859, 5905
- _regex_mode_quit_c:
..... 4965, 4965, 5077, 5486
- _regex_msg_repeated:nnN
..... 6159, 6180, 6190, 8327, 8327
- _regex_multi_match:n
555, 6731, 6741, 7532, 7552, 7561, 7776
- \c_regex_no_match_regex
..... 4432, 4883, 7316
- \c_regex_outer_mode_int
..... 4860, 4928, 4938, 4947,
4955, 4968, 5026, 5042, 5859, 5905
- _regex_peek:nnTF
592, 7851, 7860, 7869, 7879, 7887, 7887
- _regex_peek_aux:nnTF
..... 7887, 7889, 7895, 7960
- _regex_peek_end:
.... 589, 591, 7853, 7862, 7915, 7915
- \l_regex_peek_false_tl
..... 7844, 7899, 7919, 7925, 7988
- _regex_peek_reinsert:N ... 591,
593, 7918, 7919, 7925, 7927, 7927, 7988
- _regex_peek_remove_end:n
.... 589, 591, 7871, 7881, 7915, 7921
- _regex_peek_replace:nnTF
..... 7942, 7950, 7957, 7957
- _regex_peek_replace_end:
..... 7960, 7962, 7962
- _regex_peek_replacement_put:n .
..... 7968, 8005, 8005
- _regex_peek_replacement_put_-
submatch_aux:n .. 7970, 8016, 8016
- _regex_peek_replacement_-
token:n 594, 7972, 8014, 8014
- _regex_peek_replacement_var:N .
..... 7973, 8030, 8030
- \l_regex_peek_true_tl
... 593, 7844, 7898, 7918, 7924, 7977
- _regex_pop_lr_states:
..... 6277, 6306, 6314, 6321, 6414
- _regex_posix_alnum: ... 4603, 4603

```

\\_regex_posix_alpha: .....
..... 535, 4603, 4604, 4605
\\_regex_posix_ascii: ... 4603, 4607
\\_regex_posix_blank: ... 4603, 4613
\\_regex_posix_cntrl: ... 4603, 4614
\\_regex_posix_digit: .....
..... 4603, 4604, 4621, 4645
\\_regex_posix_graph: ... 4603, 4622
\\_regex_posix_lower: 4603, 4606, 4624
\\_regex_posix_print: ... 4603, 4626
\\_regex_posix_punct: ... 4603, 4628
\\_regex_posix_space: ... 4603, 4635
\\_regex_posix_upper: 4603, 4606, 4640
\\_regex_posix_word: .... 4603, 4642
\\_regex_posix_xdigit: ... 4603, 4643
\\_regex_prop_: ..... 518, 5286
\\_regex_prop_d: .....
..... 518, 535, 4574, 4574, 4621
\\_regex_prop_h: .... 4574, 4576, 4613
\\_regex_prop_N: .... 4574, 4598, 5314
\\_regex_prop_s: ..... 4574, 4581
\\_regex_prop_v: ..... 4574, 4589
\\_regex_prop_w: .....
.. 4574, 4591, 4642, 6592, 6594, 6595
\\_regex_push_lr_states: .....
..... 6268, 6304, 6314, 6314, 6412
\\_regex_quark_if_nil:N ..... 4449
\\_regex_quark_if_nil:NTF 5739, 5759
\\_regex_quark_if_nil:nTF .... 4449
\\_regex_quark_if_nil_p:n .... 4449
\\_regex_query_range:nn .....
..... 564, 593, 6914, 6920,
6920, 6939, 7010, 7760, 7797, 7979
\\_regex_query_range_loop:ww ...
..... 6920, 6922, 6927, 6934
\\_regex_query_set:n ..... 7480,
7480, 7546, 7555, 7581, 7753, 7779
\\_regex_query_set_aux:nN .....
..... 7480, 7484, 7486, 7487, 7490
\\_regex_query_set_from_input_-
tl: ..... 7967, 7990, 7990
\\_regex_query_set_item:n .....
..... 7990, 7994, 7995, 7997, 8000
\\_regex_query_submatch:n .....
.. 6937, 6937, 7124, 7657, 8020, 8025
\\_regex_reinsert_item:n .....
592, 593, 7927, 7930, 7933, 7971, 8009
\\_regex_replace_all:nnN .....
..... 7410, 7771, 7771
\\_regex_replace_all_aux:nnN ...
..... 7446, 7772, 7773
\\_regex_replace_once:nnN .....
..... 7408, 7742, 7742
\\_regex_replace_once_aux:nnN ...
..... 7423, 7742, 7743, 7744
\\_regex_replacement:n .....
..... 593, 6961, 6961, 7005, 7425,
7743, 7772, 7974, 42484, 42485, 42486
\\_regex_replacement_apply:Nn ...
..... 6961, 6962, 6963, 7040
\\_regex_replacement_balance_-
one_match:n .....
... 563, 6910, 6910, 7022, 7756, 7788
\\_regex_replacement_c:w . 7163, 7163
\\_regex_replacement_c_A:w .....
..... 567, 7095, 7251, 7252
\\_regex_replacement_c_B:w .....
..... 7083, 7254, 7255
\\_regex_replacement_c_C:w 7263, 7263
\\_regex_replacement_c_D:w .....
..... 7090, 7268, 7269
\\_regex_replacement_c_E:w .....
..... 7084, 7271, 7272
\\_regex_replacement_c_L:w .....
..... 7093, 7280, 7281
\\_regex_replacement_c_M:w .....
..... 7085, 7283, 7284
\\_regex_replacement_c_O:w 7082,
7087, 7091, 7094, 7096, 7286, 7287
\\_regex_replacement_c_P:w .....
..... 7088, 7289, 7290
\\_regex_replacement_c_S:w .....
..... 7078, 7092, 7295, 7295
\\_regex_replacement_c_T:w .....
..... 7086, 7303, 7304
\\_regex_replacement_c_U:w .....
..... 7089, 7306, 7307
\\_regex_replacement_cat:NNN ...
..... 7168, 7211, 7211
\\l__regex_replacement_category_-
seq 6907, 6992, 6995, 6996, 7065, 7225
\\l__regex_replacement_category_-
tl ..... 567,
6907, 7060, 7066, 7069, 7226, 7227
\\_regex_replacement_char:nNN ...
..... 574,
7246, 7246, 7253, 7260, 7270, 7277,
7282, 7285, 7288, 7292, 7305, 7308
\\l__regex_replacement_csnames_-
int 562, 6906, 6986, 6988, 6990,
7057, 7125, 7179, 7186, 7196, 7198,
7205, 7216, 7257, 7274, 8007, 8018
\\_regex_replacement_cu_aux:Nw ..
..... 7173, 7177, 7177, 7191
\\_regex_replacement_do_one_-
match:n ..... 593,
594, 6912, 6912, 7008, 7759, 7796, 7978

```

__regex_replacement_error:NNN 7134, 7146,
 7157, 7169, 7174, 7192, 7310, 7310
 __regex_replacement_escaped:N 6982, 7101, 7101, 7230
 __regex_replacement_exp_not:N 570, 6918, 6918, 7173, 7266, 7972
 __regex_replacement_exp_not:n 6919, 6919, 7191, 7973
 __regex_replacement_g:w 7130, 7130
 __regex_replacement_g_digits:NN 7130, 7133, 7136, 7143
 __regex_replacement_lbrace:N 6975, 7132, 7172, 7190, 7203, 7203
 __regex_replacement_normal:n 6977, 6983, 7055, 7055,
 7108, 7138, 7165, 7200, 7208, 7223
 __regex_replacement_normal_-
 aux:N 7055, 7061, 7075
 __regex_replacement_put:n 7053, 7053, 7058, 7249, 7301, 7968
 __regex_replacement_put_-
 submatch:n 7106, 7112, 7112, 7153
 __regex_replacement_put_-
 submatch_aux:n 7112, 7115, 7121, 7969
 __regex_replacement_rbrace:N 6972, 7152, 7194, 7194
 __regex_replacement_set:n 6961, 6962, 7006, 7043
 \l__regex_replacement_tl 7846, 7959, 7974
 __regex_replacement_u:w 7188, 7188
 __regex_return: 577, 7362, 7368, 7396, 7398, 7472, 7472
 \l__regex_right_state_int 6221, 6245, 6286, 6287, 6307,
 6319, 6326, 6335, 6336, 6375, 6382,
 6388, 6401, 6409, 6459, 6463, 6474,
 6488, 6497, 6509, 6513, 6517, 6520,
 6525, 6528, 6531, 6539, 6553, 6556,
 6559, 6562, 6566, 6582, 6625, 42443
 \l__regex_right_state_seq 6221, 6285, 6295, 6318, 6325, 6461
 \l__regex_saved_success_bool 555, 4563, 4570, 6655
 __regex_sep: 558, 4358, 4358, 4708, 4723,
 4742, 4748, 4753, 4754, 4763, 4771,
 5372, 5383, 6789, 6800, 6873, 6875,
 6923, 6924, 6927, 6935, 7646, 7648
 __regex_show:N 575, 6060, 6060, 7341, 7353
 __regex_show:NN 7336, 7346, 7347, 7348
 __regex_show:Nn 7336, 7336, 7337, 7338
 __regex_show_char:n 6092,
 6096, 6099, 6103, 6112, 6125, 6125
 __regex_show_class:NnnnN 6079, 6161, 6161
 __regex_show_group_aux:nnnnN 6074, 6076, 6078, 6152, 6152
 __regex_show_item_catcode:NnTF 6106, 6108, 6197, 6197
 __regex_show_item_exact_cs:n 6113, 6210, 6210
 \l__regex_show_lines_int 4885, 6133, 6165, 6168, 6175
 __regex_show_one:n 6068, 6081, 6084, 6092, 6095,
 6099, 6102, 6112, 6116, 6131, 6131,
 6147, 6154, 6158, 6171, 6187, 6215
 __regex_show_pop: 6141, 6143, 6150, 6157
 \l__regex_show_prefix_seq 4884,
 6066, 6069, 6117, 6137, 6142, 6144
 __regex_show_push:n 6118, 6141, 6141, 6148, 6155, 6166
 __regex_show_scope:nn 6110, 6115, 6141, 6145, 6202
 __regex_single_match: 555, 4560,
 6731, 6731, 7505, 7542, 7747, 7900
 __regex_split:nnN 7412, 7558, 7558
 __regex_standard_escapechar: 4359, 4359, 4656, 5025, 6237, 6258
 \l__regex_start_pos_int 6611, 6630, 6706, 6711, 6717,
 7564, 7576, 7589, 7592, 7715, 7798
 \g__regex_state_active_intarray 490, 543, 554-
 556, 6647, 6688, 6793, 6796, 6804, 6831
 \l__regex_step_int 490, 6644, 6690,
 6755, 6794, 6798, 6806, 6820, 6822
 __regex_store_state:n 554, 6716, 6845, 6848, 6848
 __regex_store_submatches: 6848
 __regex_store_submatches:n 6867
 __regex_store_submatches:nn 6850, 6854
 __regex_submatch_balance:n 6911, 6943, 6943, 7025, 7127, 7646
 \g__regex_submatch_begin_-
 intarray 490, 563, 586, 6916, 6940, 6956,
 7017, 7465, 7571, 7574, 7587, 7728
 \g__regex_submatch_case_intarray 7036, 7465, 7710, 7716

\g__regex_submatch_end_intarray .
 [490](#), [586](#), [6941](#), [6949](#),
 [7465](#), [7568](#), [7584](#), [7731](#), [7762](#), [7981](#)
 \l__regex_submatch_int
 [490](#), [579](#), [582](#), [583](#), [587](#), [6697](#), [7462](#),
 [7583](#), [7585](#), [7588](#), [7590](#), [7593](#), [7606](#),
 [7705](#), [7709](#), [7711](#), [7712](#), [7787](#), [7795](#)
 \g__regex_submatch_prev_intarray
 [490](#), [579](#), [586](#), [6915](#),
 [7013](#), [7465](#), [7566](#), [7582](#), [7708](#), [7714](#)
 \g__regex_success_bool
 [555](#), [4564](#), [4566](#), [4569](#), [6655](#),
 [6683](#), [6736](#), [6748](#), [7427](#), [7450](#), [7474](#),
 [7523](#), [7704](#), [7750](#), [7917](#), [7923](#), [7964](#)
 \l__regex_success_pos_int
 [6630](#), [6692](#), [6711](#), [6898](#), [7564](#)
 \l__regex_success_submatches_tl .
 [554](#), [586](#), [6642](#), [6901](#), [7719](#)
 __regex_tests_action_cost:n ...
 .. [6353](#), [6355](#), [6368](#), [6374](#), [6383](#), [6401](#)
 \g__regex_thread_info_intarray ..
 . [490](#), [553-555](#), [561](#), [6647](#), [6786](#), [6857](#)
 __regex_tl_even_items:n
 [4410](#), [4410](#), [4411](#), [7448](#)
 __regex_tl_even_items_loop:nn ..
 [4410](#), [4413](#), [4416](#), [4420](#)
 __regex_tl_odd_items:n
 [4410](#), [4410](#), [7424](#), [7447](#), [7521](#)
 __regex_tmp:w [583](#), [4422](#), [4422](#), [5298](#),
 [5308](#), [5309](#), [5310](#), [5311](#), [5312](#), [5335](#),
 [5346](#), [5347](#), [7391](#), [7404](#), [7406](#), [7408](#),
 [7410](#), [7412](#), [7602](#), [7607](#), [7630](#), [7637](#),
 [7644](#), [7682](#), [7687](#), [7691](#), [7695](#), [7700](#)
 \l__regex_tmp_bool
 [4423](#), [5436](#), [5441](#), [5462](#), [5471](#)
 \l__regex_tmp_regex
 [510](#), [4883](#), [5020](#), [5058](#), [5715](#),
 [5721](#), [6233](#), [7324](#), [7329](#), [7334](#), [7341](#)
 \l__regex_tmp_seq [4423](#), [6199](#),
 [6200](#), [6205](#), [6212](#), [6213](#), [6214](#), [6216](#)
 \g__regex_tmp_tl .. [583](#), [584](#), [4423](#),
 [4657](#), [4661](#), [5867](#), [5874](#), [7603](#), [7614](#),
 [7615](#), [7633](#), [7678](#), [7681](#), [7817](#), [7822](#)
 \l__regex_tmpa_int [515](#), [569](#),
 [4423](#), [5154](#), [5165](#), [5176](#), [5185](#), [5189](#),
 [5197](#), [5200](#), [5204](#), [5207](#), [5214](#), [6386](#),
 [6389](#), [6395](#), [6400](#), [6474](#), [6489](#), [6495](#),
 [6501](#), [6510](#), [6513](#), [6517](#), [6520](#), [6525](#),
 [6528](#), [6531](#), [6546](#), [6554](#), [6563](#), [7133](#),
 [7154](#), [7718](#), [7727](#), [7729](#), [7734](#), [7739](#)
 \l__regex_tmpa_tl [499](#), [531](#),
 [532](#), [536](#), [588](#), [4423](#), [4547](#), [4550](#),
 [4652](#), [4659](#), [4666](#), [5437](#), [5442](#), [5458](#),
 [5463](#), [5468](#), [5472](#), [5478](#), [5479](#), [5713](#),
 [5724](#), [5782](#), [5826](#), [5858](#), [5870](#), [5886](#),
 [6067](#), [6070](#), [6123](#), [6144](#), [6186](#), [6193](#),
 [6285](#), [6286](#), [6323](#), [6324](#), [6325](#), [6326](#),
 [6456](#), [6457](#), [6461](#), [6463](#), [6719](#), [6722](#),
 [7344](#), [7356](#), [7757](#), [7790](#), [7825](#), [42417](#)
 \l__regex_tmpb_int ... [4423](#), [5169](#),
 [5198](#), [5201](#), [5202](#), [5204](#), [5208](#), [5215](#),
 [6490](#), [6495](#), [6500](#), [6546](#), [6554](#), [6563](#)
 \l__regex_tmpb_tl
 [4423](#), [5781](#), [5801](#), [5814](#)
 \l__regex_tmpc_int
 [4423](#), [6492](#), [6497](#), [6498](#), [6502](#)
 __regex_toks_clear:N
 [4362](#), [4362](#), [6294](#), [6334](#)
 __regex_toks_memcpy:NNn
 [4367](#), [4367](#), [6499](#)
 __regex_toks_put_left:Nn
 [4376](#), [4377](#),
 [4379](#), [6266](#), [6287](#), [6329](#), [6481](#), [6482](#)
 __regex_toks_put_right:Nn
 [491](#), [4376](#),
 [4383](#), [4385](#), [4389](#), [4391](#), [6242](#), [6245](#),
 [6262](#), [6307](#), [6331](#), [6342](#), [6573](#), [6618](#)
 __regex_toks_set:Nn
 [4362](#), [4364](#), [4365](#), [7493](#), [8003](#)
 __regex_toks_use:w
 [4361](#), [4361](#), [6795](#), [6933](#), [8361](#)
 __regex_trace:nnn
 ... [8343](#), [8344](#), [8346](#), [8347](#), [8360](#),
 [42439](#), [42461](#), [42468](#), [42473](#), [42478](#)
 __regex_trace_pop:nnN
 . [8343](#), [8345](#), [42418](#), [42427](#), [42434](#),
 [42452](#), [42456](#), [42463](#), [42470](#), [42485](#)
 __regex_trace_push:nnN
 . [8343](#), [8343](#), [42415](#), [42424](#), [42431](#),
 [42451](#), [42455](#), [42460](#), [42467](#), [42484](#)
 \g__regex_trace_regex_int [8353](#)
 __regex_trace_states:n
 [8354](#), [8354](#), [42426](#), [42433](#)
 __regex_two_if_eq:NNNTF
 [4886](#), [4886](#), [5135](#), [5182](#), [5195](#), [5229](#),
 [5397](#), [5434](#), [5454](#), [5455](#), [5524](#), [5559](#),
 [5576](#), [5577](#), [5639](#), [5772](#), [5779](#), [7223](#)
 __regex_use_i_delimit_by_q-
 recursion_stop:nw [4444](#), [4446](#), [5762](#)
 __regex_use_none_delimit_by_q-
 nil:w [4418](#), [4444](#), [4448](#)
 __regex_use_none_delimit_by_q-
 recursion_stop:w
 [4444](#), [4444](#), [5740](#), [5764](#), [7720](#)
 __regex_use_state:
 [6791](#), [6791](#), [6808](#), [6834](#), [42482](#)
 __regex_use_state_and_submatches:w
 [558](#), [6784](#), [6800](#), [6800](#)

- _regex_Z_test: 505, 5328,
 5330, 5347, 5967, 6088, 6570, 6597
 \l_regex_zeroth_submatch_int . . .
 579, 585, 7462, 7567, 7569,
 7572, 7575, 7705, 7715, 7717, 7729,
 7734, 7756, 7759, 7763, 7978, 7982
 register commands:
 register_lua_data 12142
 \relax . . . 4, 8, 13, 17, 53, 54, 61, 85, 86,
 87, 88, 89, 90, 91, 92, 93, 94, 96, 97,
 98, 99, 100, 101, 102, 103, 104, 105, 383
 \relpenalty 384
 \resettimer 776
 reverse commands:
 \reverse_if:N
 29, 712, 773, 887, 888, 1106, 1386,
 1391, 4466, 4467, 4484, 4485, 4490,
 4491, 8580, 12216, 14271, 18296,
 18471, 18473, 18475, 18477, 18564,
 21728, 21733, 21737, 21739, 25136,
 28832, 29673, 29706, 32495, 32532
 \right 385
 \rightghost 911
 \righthyphenmin 386
 \rightmarginkern 679
 \rightskip 387
 \romannumeral 388
 round 283
 \rpostcode 680

 S
 \saveboxresource 958
 \savecatcodetable 912
 \saveimageresource 959
 \savepos 957
 \savingshyphcodes 519
 \savingsvdiscards 520
 scan commands:
 \scan_new:N 155, 753,
 855, 3194, 3195, 3604, 8625, 8626,
 9309, 9310, 10534, 10535, 11040,
 13294, 13573, 13574, 13575, 13683,
 13684, 14457, 17344, 17344, 17371,
 17372, 17373, 18204, 18205, 19182,
 19183, 19865, 20091, 20092, 20544,
 20545, 20741, 20742, 20747, 21583,
 21584, 22056, 22220, 22221, 22222,
 22223, 22488, 22489, 22490, 24231,
 24234, 24235, 24236, 24237, 24239,
 24240, 24241, 24242, 24243, 24342,
 30628, 32663, 32672, 32673, 38631,
 38645, 40432, 40872, 41680, 42357
 \scan_stop: 14, 24, 25,
 154, 155, 172, 213, 384, 387, 406,
 407, 410, 420, 427, 482, 483, 497,
 505, 528, 604, 686, 715, 720, 730–
 732, 737, 744, 773, 854, 888, 894,
 942, 951, 953, 954, 956, 974, 978,
 985, 986, 992, 1102, 1106–1108,
 1111, 1352, 1442, 1566, 121, 134,
 1415, 1415, 1826, 1848, 1864, 1872,
 1892, 1908, 1943, 1956, 2320, 2344,
 2355, 2364, 2375, 2384, 2395, 2487,
 2753, 2754, 2769, 2809, 2835, 2859,
 2876, 3066, 3072, 3228, 3609, 3775,
 3815, 3819, 3825, 3827, 3874, 3876,
 4169, 4178, 4180, 4190, 4231, 4232,
 4233, 4239, 4527, 4548, 4549, 4662,
 4722, 4747, 4759, 4791, 4905, 5673,
 5732, 6050, 6054, 6057, 6212, 6798,
 6810, 6959, 7124, 7127, 7248, 7300,
 7602, 8020, 8025, 8981, 8985, 9198,
 9210, 9222, 9281, 10324, 10329,
 10451, 10575, 10578, 11155, 11162,
 11194, 11527, 12490, 12639, 12788,
 12798, 12861, 12891, 12893, 13249,
 13269, 13283, 14272, 14567, 15559,
 17066, 17353, 17356, 17842, 18700,
 20076, 20184, 20253, 20353, 20593,
 20654, 20730, 20733, 20735, 21150,
 21298, 21595, 21614, 21616, 21620,
 21623, 21626, 21630, 21635, 21639,
 21887, 22054, 22066, 22084, 22086,
 22094, 22096, 22100, 22102, 22124,
 22129, 22132, 22158, 22178, 22180,
 22188, 22190, 22194, 22196, 22200,
 23796, 23879, 23978, 24016, 24023,
 24183, 24214, 24406, 25134, 25138,
 25339, 25356, 25658, 25705, 25706,
 25965, 26008, 26036, 26050, 26879,
 28741, 28749, 29498, 29501, 29504,
 29507, 29510, 29513, 29516, 29519,
 29522, 30597, 30620, 30860, 31001,
 31565, 31592, 31801, 32690, 32691,
 36391, 36537, 37164, 37167, 38414,
 41379, 41382, 41869, 41892, 41905,
 41987, 42012, 42074, 42083, 42499
 \s_stop 5, 155, 855, 17356, 17367
 scan internal commands:
 \s__bool_mark 8625, 8638, 8646
 \s__bool_stop 8625, 8638, 8646
 \s__char_stop 19865
 \s__clist_mark 915, 917–919,
 924, 19182, 19184, 19212, 19213,
 19230, 19359, 19369, 19373, 19395,
 19445, 19451, 19465, 19477, 19478,
 19479, 19482, 19483, 19484, 19493,
 19494, 19503, 19701, 19702, 19714,

- 19715, 19730, 19738, 19744, 19747
- \s__clist_stop 918, 920, 924, 19182, 19185, 19186, 19198, 19202, 19344, 19347, 19359, 19362, 19370, 19373, 19381, 19395, 19451, 19479, 19482, 19483, 19495, 19503, 19546, 19547, 19554, 19558, 19560, 19562, 19569, 19575, 19591, 19592, 19618, 19619, 19626, 19631, 19633, 19635, 19641, 19648, 19676, 19681, 19703, 19714, 19715, 19716, 19731, 19744, 19747, 19777, 19812
- \s__color_mark 38645, 38904, 38906, 38909, 38916, 39212, 39217, 39223, 39226, 39233, 39445, 39448, 39458, 39875, 39917, 39920, 39938, 39944, 40060, 40089, 40092, 40106, 40117, 40121, 40124, 40132, 40138, 40142, 40145, 40158, 40168
- \s__color_stop 1482, 38631, 38674, 38680, 38681, 38688, 38692, 38693, 38696, 38702, 38704, 38706, 38708, 38710, 38712, 38731, 38733, 38739, 38759, 38788, 38805, 38811, 38828, 38904, 38906, 38909, 38916, 38923, 38925, 38934, 38945, 38946, 38948, 38950, 38952, 38992, 38999, 39000, 39001, 39010, 39019, 39084, 39089, 39117, 39155, 39157, 39213, 39217, 39223, 39226, 39233, 39241, 39396, 39399, 39433, 39439, 39445, 39448, 39458, 39527, 39531, 39555, 39557, 39559, 39561, 39579, 39694, 39707, 39711, 39722, 39729, 39737, 39743, 39751, 39753, 39759, 39763, 39764, 39785, 39787, 39866, 39872, 39875, 39883, 39897, 39911, 39914, 39917, 39921, 39924, 39934, 39938, 39944, 39971, 40000, 40055, 40056, 40063, 40074, 40075, 40089, 40092, 40106, 40117, 40121, 40124, 40132, 40138, 40142, 40145, 40158, 40168, 40212, 40213
- \s__cs_mark 406, 435, 437, 1814, 1815, 1818, 1819, 1820, 2753, 2783, 2784, 2786, 2792, 2796, 2818, 2827, 2846, 2874, 2877, 2885, 2900, 2932, 2946, 2950, 2959, 2978, 2987, 2992, 3067, 3070, 3086, 17363
- \s__cs_stop 406, 437, 1815, 1818, 1819, 1820, 2334, 2353, 2373, 2393, 2753, 2756, 2757, 2787, 2796, 2822, 2874, 2877, 2881, 2889, 2895, 2904, 2910, 2912, 2932, 2954, 2959, 2989, 2992, 3067, 17364
- \s__debug_stop 41680, 41681, 41855, 41857, 42048, 42062
- \s__dim_mark 21583, 21769, 21776
- \s__dim_stop 21583, 21585, 21716, 21740, 21769, 21776
- \s__file_stop .. 695, 11013, 11018, 11040, 11108, 11109, 11113, 11120, 11122, 11123, 11248, 11249, 11254, 11256, 11258, 11587, 11589, 11592, 11593, 11595, 11607, 11683, 11686, 11693, 11695, 11711, 11712, 11715
- \s__fp 1063–1065, 1070, 1071, 1096, 1102, 1104, 1106, 1120, 1122, 1123, 1155, 1159, 1161, 1163, 1169, 1172, 1265, 24231, 24244, 24245, 24246, 24247, 24248, 24258, 24263, 24265, 24266, 24281, 24294, 24297, 24299, 24309, 24321, 24341, 24358, 24361, 24368, 24375, 24391, 24418, 24525, 24527, 24529, 24530, 24531, 24534, 24535, 24536, 24538, 24554, 24720, 24725, 24963, 25019, 25028, 25030, 25707, 25862, 26348, 26363, 26387, 26407, 26408, 26504, 26519, 26521, 26539, 26544, 26545, 26607, 26643, 26644, 26658, 26659, 26696, 26697, 26800, 26801, 26802, 26813, 26829, 26835, 26901, 26902, 26905, 26916, 26917, 26925, 26926, 26929, 26930, 26931, 26933, 26934, 26935, 26947, 26950, 26954, 26957, 26978, 27028, 27031, 27034, 27054, 27055, 27057, 27058, 27059, 27067, 27070, 27081, 27082, 27084, 27093, 27170, 27326, 27360, 27361, 27364, 27365, 27448, 27449, 27589, 27597, 27599, 27776, 27785, 27787, 27792, 27800, 27802, 27804, 27807, 28336, 28348, 28350, 28569, 28586, 28588, 28771, 28790, 28792, 28793, 28796, 28813, 28816, 28819, 28843, 28844, 28846, 28863, 28953, 28966, 28968, 28971, 28976, 29009, 29025, 29108, 29121, 29123, 29136, 29138, 29151, 29153, 29166, 29168, 29181, 29183, 29196, 29206, 29734, 29750, 29751, 29755, 29766, 29874, 29887, 29889, 29905, 29908, 29918, 29941, 29952, 29954, 29968, 29970, 29975, 30040, 30061, 30064, 30094, 30115, 30118, 30168, 30184, 30187, 30262, 30263, 30347, 30349, 30381, 31205, 31213, 31216, 31296
- \s__fp_⟨type⟩ 1096

- \s__fp_division [24239](#)
- \s__fp_exact [24239](#), [24244](#),
[24245](#), [24246](#), [24247](#), [24248](#), [26901](#)
- \s__fp_expr_mark
... [1102](#), [1103](#), [1106](#), [1128](#), [1132](#),
[24234](#), [25913](#), [25928](#), [26009](#), [26051](#)
- \s__fp_expr_stop
[1072](#), [24234](#), [24432](#), [25813](#), [25914](#),
[25918](#), [25929](#), [26985](#), [26996](#), [27006](#),
[27014](#), [30656](#), [30816](#), [31018](#), [31107](#)
- \s__fp_invalid [24239](#)
- \s__fp_mark
[24236](#), [24381](#), [24382](#), [24386](#), [30571](#),
[30573](#), [30581](#), [30640](#), [30641](#), [30645](#)
- \s__fp_overflow [24239](#), [24265](#)
- \s__fp_stop
... [1070](#), [24236](#), [24238](#), [24282](#),
[24358](#), [24369](#), [24376](#), [24382](#), [24386](#),
[24400](#), [24419](#), [25234](#), [25238](#), [25743](#),
[25748](#), [26348](#), [26370](#), [26522](#), [26527](#),
[26532](#), [26539](#), [26550](#), [26559](#), [26606](#),
[26607](#), [26643](#), [26644](#), [26800](#), [26801](#),
[26802](#), [26977](#), [26978](#), [28645](#), [28660](#),
[30013](#), [30018](#), [30575](#), [30642](#), [30645](#)
- \s__fp_symbolic [1280–1282](#), [26506](#),
[26529](#), [26532](#), [26559](#), [26563](#), [30628](#),
[30634](#), [30641](#), [30645](#), [30647](#), [30665](#),
[30678](#), [30698](#), [30727](#), [30750](#), [30822](#),
[30826](#), [30843](#), [31011](#), [31052](#), [31061](#)
- \s__fp_tuple [1069](#),
[24342](#), [24348](#), [24349](#), [24426](#), [24428](#),
[26128](#), [26340](#), [26355](#), [26380](#), [26382](#),
[26399](#), [26400](#), [26402](#), [26505](#), [26524](#),
[26527](#), [26550](#), [26554](#), [26688](#), [26689](#),
[27845](#), [27846](#), [27852](#), [27853](#), [29987](#)
- \s__fp_underflow [24239](#), [24263](#)
- \s__graphics_stop
... [40432](#), [40547](#), [40556](#), [40561](#),
[40564](#), [40574](#), [40577](#), [40774](#), [40784](#)
- \s__int_mark
.. [18204](#), [18444](#), [18447](#), [18545](#), [18552](#)
- \s__int_stop [888](#), [900](#), [18204](#), [18206](#),
[18423](#), [18439](#), [18441](#), [18445](#), [18458](#),
[18545](#), [18552](#), [18964](#), [18970](#), [18987](#)
- \s__iow_mark . [10534](#), [10903](#), [10910](#),
[10922](#), [10996](#), [10997](#), [10998](#), [10999](#)
- \s__iow_stop
... [10534](#), [10536](#), [10788](#), [10830](#),
[10888](#), [10926](#), [10939](#), [10996](#), [10999](#)
- \s__keys_mark [22488](#), [22561](#), [22564](#),
[22576](#), [22578](#), [22582](#), [23301](#), [23304](#),
[23309](#), [23315](#), [23319](#), [23324](#), [23594](#),
[23597](#), [23606](#), [23608](#), [23613](#), [23616](#)
- \s__keys_nil [22488](#), [22556](#),
[22557](#), [22559](#), [22561](#), [22564](#), [22573](#),
[22574](#), [22576](#), [22578](#), [22581](#), [22582](#),
[22589](#), [23296](#), [23297](#), [23299](#), [23301](#),
[23304](#), [23307](#), [23315](#), [23316](#), [23593](#),
[23596](#), [23602](#), [23604](#), [23612](#), [23615](#)
- \s__keys_stop [22488](#), [22613](#),
[22618](#), [22756](#), [22804](#), [22909](#), [22918](#),
[22922](#), [22924](#), [23278](#), [23294](#), [23526](#),
[23546](#), [23697](#), [23704](#), [23709](#), [23714](#)
- \s__keyval_mark [1007–](#)
[1009](#), [1012](#), [22220](#), [22234](#), [22245](#),
[22246](#), [22247](#), [22248](#), [22254](#), [22255](#),
[22257](#), [22262](#), [22263](#), [22266](#), [22267](#),
[22268](#), [22273](#), [22274](#), [22278](#), [22279](#),
[22282](#), [22283](#), [22286](#), [22287](#), [22290](#),
[22293](#), [22294](#), [22299](#), [22302](#), [22303](#),
[22307](#), [22308](#), [22311](#), [22314](#), [22318](#),
[22319](#), [22320](#), [22321](#), [22328](#), [22329](#),
[22338](#), [22339](#), [22341](#), [22345](#), [22359](#),
[22360](#), [22368](#), [22369](#), [22378](#), [22379](#),
[22380](#), [22381](#), [22383](#), [22404](#), [22405](#),
[22410](#), [22414](#), [22416](#), [22418](#), [22430](#)
- \s__keyval_nil [1008](#),
[22220](#), [22253](#), [22261](#), [22266](#), [22268](#),
[22269](#), [22270](#), [22272](#), [22278](#), [22281](#),
[22286](#), [22290](#), [22292](#), [22299](#), [22301](#),
[22307](#), [22311](#), [22313](#), [22318](#), [22320](#),
[22328](#), [22339](#), [22359](#), [22368](#), [22403](#),
[22407](#), [22423](#), [22426](#), [22430](#), [22431](#)
- \s__keyval_stop ... [22220](#), [22246](#),
[22248](#), [22259](#), [22267](#), [22279](#), [22287](#),
[22290](#), [22296](#), [22308](#), [22311](#), [22313](#),
[22314](#), [22318](#), [22328](#), [22359](#), [22360](#),
[22368](#), [22369](#), [22378](#), [22381](#), [22383](#)
- \s__keyval_tail [1008](#),
[22220](#), [22234](#), [22242](#), [22243](#), [22252](#),
[22336](#), [22338](#), [22344](#), [22345](#), [22380](#)
- \s__msg_mark
.. [9309](#), [9646](#), [9729](#), [9730](#), [9735](#), [9738](#)
- \s__msg_stop [9309](#),
[9311](#), [9648](#), [9652](#), [9654](#), [9731](#), [10215](#)
- \s__pdf_stop . [40872](#), [41072](#), [41073](#),
[41081](#), [41093](#), [41106](#), [41110](#), [41112](#)
- \s__peek_mark
..... [20544](#), [20707](#), [20708](#), [20715](#)
- \s__peek_stop
.. [20544](#), [20546](#), [20696](#), [20709](#), [20718](#)
- \s__prg_mark [1661](#), [1663](#), [1671](#)
- \s__prg_stop [1688](#), [1693](#), [1712](#), [1720](#),
[1728](#), [1784](#), [1788](#), [1790](#), [1792](#), [1794](#)
- \s__prop [956–](#)
[958](#), [963](#), [967](#), [975](#), [977](#), [978](#), [980–](#)
[982](#), [20747](#), [20755](#), [20757](#), [20758](#),

- 20762, 20765, 20767, 20769, 20772,
20803, 20822, 20845, 20848, 20854,
20857, 20896, 20906, 20909, 20912,
20915, 20918, 20922, 20986, 21043,
21046, 21051, 21090, 21099, 21103,
21126, 21137, 21157, 21172, 21173,
21276, 21282, 21290, 21321, 21340,
21344, 21381, 21382, 21383, 21384,
21388, 21389, 21390, 21391, 21405,
21419, 21420, 21421, 21422, 21426,
21427, 21428, 21429, 21480, 21482,
21483, 21486, 21497, 21500, 21503,
21507, 21518, 21539, 21570, 21571
- \s__prop_mark
.. 967, 968, 20741, 20798, 20799,
20802, 21043, 21045, 21047, 21098,
21099, 21103, 21523, 21542, 21543
- \s__prop_stop 967,
20741, 20799, 20802, 21043, 21048,
21056, 21057, 21100, 21103, 21340,
21344, 21486, 21503, 21523, 21543
- \s__quark
17066, 17315, 17317, 17318, 17329,
17332, 17337, 17340, 17342, 17361
- \g__scan_marks_tl
..... 855, 17346, 17352, 17356
- \s__seq
856, 861, 864, 869, 873, 875, 877,
17371, 17382, 17412, 17417, 17422,
17427, 17438, 17470, 17549, 17557,
17561, 17665, 17677, 17679, 17844,
17892, 18047, 18054, 18136, 18175
- \s__seq_mark
.. 17372, 18124, 18125, 18139, 18142
- \s__seq_stop
17372, 17668, 17679, 17797, 17800,
17808, 17810, 17891, 17892, 18046,
18047, 18049, 18054, 18058, 18060,
18065, 18126, 18139, 18142, 18144
- \s__skip_stop
..... 22056, 22118, 22120, 42737
- \s__sort_mark 454, 457-459,
3194, 3390, 3394, 3400, 3404, 3410,
3413, 3478, 3479, 3481, 3518, 3520,
3523, 3527, 3530, 3533, 3535, 3538
- \s__sort_stop 456, 458, 459, 3194,
3466, 3475, 3479, 3481, 3518, 3519,
3520, 3525, 3527, 3531, 3533, 3541
- \s__str 785,
793, 811, 814, 14457, 14606, 14610,
14794, 14841, 14909, 14912, 15357,
15369, 15374, 15384, 15389, 15394,
15397, 15412, 15425, 15428, 15563,
15564, 15581, 15587, 15603, 15609,
15610, 15715, 15730, 15739, 15740
- \s__str_mark 760,
764, 767, 774, 13683, 13883, 13918,
13927, 14010, 14027, 14278, 14280
- \s__str_stop 767,
771, 810, 814, 833, 837-842, 13683,
13685, 13686, 13790, 13883, 13918,
13927, 14010, 14019, 14025, 14027,
14033, 14050, 14070, 14132, 14189,
14201, 14241, 14257, 14264, 14272,
14274, 14278, 14280, 14606, 14612,
14654, 14659, 14669, 14864, 14867,
14886, 14892, 15271, 15274, 15282,
15370, 15406, 15520, 15522, 15526,
15538, 15678, 15680, 15684, 15696,
15705, 15712, 15733, 16636, 16655,
16700, 16704, 16801, 16881, 16883,
16938, 16941, 16983, 17037, 17041
- \s__text_recursion_stop .. 32672,
32675, 32999, 33013, 33022, 33064,
33073, 33217, 33225, 33304, 33312
- \s__text_recursion_tail
..... 32672, 32679, 32680, 32999
- \s__text_stop
.. 32663, 32758, 32760, 33238, 33239
- \s__tl 463-466,
474, 475, 3603, 3604, 3872, 3908,
3914, 3940, 3958, 3967, 3984, 3988,
4023, 4026, 4163, 4167, 4208, 4212
- \s__tl_act_stop . 745, 746, 13294,
13300, 13301, 13304, 13307, 13311,
13320, 13323, 13326, 13329, 13332,
13334, 13336, 13340, 13343, 13349
- \s__tl_mark 13046,
13047, 13050, 13053, 13054, 13573
- \s__tl_nil 739, 13100,
13109, 13118, 13119, 13131, 13133,
13134, 13136, 13137, 13141, 13573
- \s__tl_stop 722, 734, 738,
12552, 12554, 12597, 12600, 12603,
12607, 12893, 12899, 12931, 12932,
12942, 12946, 12948, 12950, 12952,
12961, 12962, 12972, 12973, 12982,
12987, 12989, 12991, 13048, 13050,
13055, 13057, 13160, 13175, 13189,
13215, 13240, 13537, 13547, 13573
- \s__token_mark
.... 949, 20091, 20523, 20524, 20533
- \s__token_stop .. 942, 945, 20091,
20241, 20244, 20274, 20309, 20445,
20449, 20455, 20478, 20525, 20533
- \scantextokens 913
- \scantokens 521
- \scriptbaselineshiftfactor 1181

- \scriptfont 389
- \scriptscriptbaselineshiftfactor . 1183
- \scriptscriptfont 390
- \scriptscriptstyle 391
- \scriptspace 392
- \scriptstyle 393
- \scrollmode 394
- sec 283
- secd 284
- \selectfont 35727
- seq commands:
 - \c_empty_seq 169, 857, 17382, 17386, 17390, 17393, 17706, 17776, 17784
 - \seq_clear:N 156, 169, 6117, 6996, 7629, 9228, 9727, 9790, 11635, 11728, 17389, 17389, 17391, 17396, 17582
 - \seq_clear_new:N 156, 17395, 17395, 17397
 - \seq_concat:NNN 159, 169, 11641, 17535, 17535, 17539, 42127
 - \seq_const_from_clist:Nn 157, 17435, 17435, 17440
 - \seq_count:N .. 160, 166, 168, 263, 6995, 11749, 17637, 17719, 17903, 17917, 18088, 18088, 18111, 18116
 - \seq_elt:w 856
 - \seq_elt_end: 856
 - \seq_format:Nn 167, 16811, 16811, 16820
 - \seq_gclear:N 156, 451, 3331, 3340, 17389, 17392, 17394, 17399, 17731, 17739
 - \seq_gclear_new:N 156, 17395, 17398, 17400
 - \seq_gconcat:NNN 159, 11654, 17535, 17537, 17540, 42128
 - \seq_get:NN 167, 6456, 6461, 18155, 18155, 18156, 18161, 18162, 39337
 - \seq_get:NNTF 167, 18161
 - \seq_get_left:NN 160, 17792, 17792, 17802, 17862, 17863, 17866, 18155, 18156, 18161, 18162
 - \seq_get_left:NNTF 161, 17862
 - \seq_get_right:NN 160, 17817, 17817, 17834, 17864, 17865, 17868
 - \seq_get_right:NNTF 161, 17862
 - \seq_gpop:NN 167, 11548, 18155, 18159, 18160, 18165, 18166, 31696, 39330
 - \seq_gpop:NNTF 168, 10313, 10564, 18161, 31666, 31678
 - \seq_gpop_left:NN 160, 17803, 17805, 17816, 17873, 17884, 18159, 18160, 18165, 18166
 - \seq_gpop_left:NNTF 161, 17870
 - \seq_gpop_right:NN 160, 17835, 17837, 17861, 17879, 17888
 - \seq_gpop_right:NNTF 162, 17870, 41349
 - \seq_gpush:Nn 33, 168, 10360, 10604, 11533, 18149, 18152, 18153, 18154, 31670, 31680, 31689, 39266
 - \seq_gput_left:Nn 159, 17545, 17553, 17564, 17565, 18152
 - \seq_gput_right:Nn 159, 3335, 11015, 11022, 11522, 17566, 17568, 17572, 17573, 17734, 40642, 41344
 - \seq_gremove_all:Nn 162, 17592, 17594, 17618, 17619
 - \seq_gremove_duplicates:N 162, 17576, 17578, 17591
 - \seq_greverse:N 163, 17686, 17688, 17703
 - \seq_gset_eq:NN 156, 3313, 17393, 17401, 17405, 17406, 17407, 17408, 17523, 17579, 17716, 42110, 42265
 - \seq_gset_filter:NNn 158, 17486, 17488
 - \seq_gset_from_clist:NN 157, 17409, 17419, 17432, 17433
 - \seq_gset_from_clist:Nn 157, 17409, 17424, 17434
 - \seq_gset_item:Nnn 162, 17621, 17623, 17626, 17629, 17632
 - \seq_gset_item:NnnTF 162, 17621
 - \seq_gset_map:NNn .. 165, 18078, 18080
 - \seq_gset_map_e:NNn 166, 18068, 18070, 41540, 41541
 - \seq_gset_map_x:NNn ... 41538, 41541
 - \seq_gset_regex_extract_all:NNn . 158, 17496
 - \seq_gset_regex_extract_all:Nnn . 158, 17496
 - \seq_gset_regex_extract_once:Nnn 158, 17496
 - \seq_gset_regex_extract_once:Nnn 158, 17496
 - \seq_gset_regex_split:NNn 159, 17496
 - \seq_gset_regex_split:Nnn 159, 17496
 - \seq_gset_split:Nnn 157, 17441, 17443, 17482, 17483
 - \seq_gset_split_keep_spaces:Nnn . 157, 17441, 17447, 17485
 - \seq_gshuffle:N 163, 17714, 17716, 17752
 - \seq_gsort:Nn 163, 3309, 3312, 3314, 17704
 - \seq_if_empty:N 17704, 17712
 - \seq_if_empty:NNTF 163, 6992, 17704, 17916, 19261, 31730

- \seq_if_empty_p:N [163](#), [17704](#)
- \seq_if_exist:N [17541](#), [17543](#)
- \seq_if_exist:NNTF
 [159](#), [17396](#), [17399](#), [17541](#), [18114](#)
- \seq_if_exist_p:N [159](#), [17541](#)
- \seq_if_in:Nn [919](#), [17753](#), [17772](#)
- \seq_if_in:NnTF [163](#),
 [168](#), [169](#), [10359](#), [10603](#), [17585](#), [17753](#)
- \seq_indexed_map_function:NN ...
 [41532](#), [41535](#)
- \seq_indexed_map_inline:Nn
 [41532](#), [41533](#)
- \seq_item:Nn .. [59](#), [160](#), [871](#), [9808](#),
 [9809](#), [9814](#), [17890](#), [17890](#), [17913](#), [17917](#)
- \seq_log:N ... [170](#), [18167](#), [18169](#), [18170](#)
- \seq_map_break:
 [158](#), [165](#), [166](#), [17920](#),
 [17920](#), [17921](#), [17923](#), [17933](#), [17974](#),
 [17984](#), [18007](#), [18015](#), [18024](#), [18052](#)
- \seq_map_break:n
 . [165](#), [872](#), [3310](#), [3313](#), [9747](#), [9761](#),
 [11206](#), [17920](#), [17922](#), [40721](#), [40739](#)
- \seq_map_function:NN
 [6](#), [90](#), [163](#), [164](#), [874](#),
 [6137](#), [6205](#), [9812](#), [11644](#), [17924](#),
 [17924](#), [17947](#), [18182](#), [19267](#), [40697](#)
- \seq_map_indexed_function:NN ...
 [164](#), [18012](#), [18012](#), [41534](#), [41535](#)
- \seq_map_indexed_inline:Nn
 [164](#), [18012](#), [18017](#), [41532](#), [41533](#)
- \seq_map_inline:Nn [163](#), [164](#),
 [169](#), [860](#), [3310](#), [3313](#), [9742](#), [17583](#),
 [17970](#), [17970](#), [17976](#), [40718](#), [40736](#)
- \seq_map_pairwise_function:NNN ..
 [164](#), [18045](#), [18045](#), [18067](#), [41536](#), [41537](#)
- \seq_map_tokens:Nn [163](#),
 [164](#), [11205](#), [11753](#), [17977](#), [17977](#), [17986](#)
- \seq_map_variable:NNn
 [164](#), [17999](#), [17999](#), [18009](#), [18010](#)
- \seq_mapthread_function:NNN
 [41536](#), [41537](#)
- \seq_new:N [6](#), [156](#),
 [3181](#), [4429](#), [4884](#), [6223](#), [6224](#), [6908](#),
 [9697](#), [9698](#), [10265](#), [10518](#), [11007](#),
 [11032](#), [11038](#), [11039](#), [17383](#), [17383](#),
 [17388](#), [17396](#), [17399](#), [17575](#), [17714](#),
 [18192](#), [18193](#), [18194](#), [18195](#), [19406](#),
 [19962](#), [19965](#), [31510](#), [31511](#), [31512](#),
 [39253](#), [40591](#), [40592](#), [40595](#), [41334](#)
- \seq_pop:NN
 [167](#), [6285](#), [6323](#), [6325](#), [7065](#),
 [18155](#), [18157](#), [18158](#), [18163](#), [18164](#)
- \seq_pop:NNTF [168](#), [18161](#)
- \seq_pop_left:NN
 . [160](#), [17803](#), [17803](#), [17815](#), [17870](#),
 [17882](#), [18157](#), [18158](#), [18163](#), [18164](#)
- \seq_pop_left:NNTF [161](#), [17870](#)
- \seq_pop_right:NN [160](#), [6066](#), [6144](#),
 [17835](#), [17835](#), [17860](#), [17876](#), [17886](#)
- \seq_pop_right:NNTF [162](#), [17870](#)
- \seq_push:Nn [168](#), [6295](#), [6316](#),
 [6318](#), [7225](#), [18149](#), [18149](#), [18150](#), [18151](#)
- \seq_put_left:Nn [159](#), [9737](#),
 [17545](#), [17545](#), [17562](#), [17563](#), [18149](#)
- \seq_put_right:Nn [159](#), [168](#),
 [169](#), [6069](#), [6142](#), [7639](#), [9798](#), [11730](#),
 [17566](#), [17566](#), [17570](#), [17571](#), [17586](#)
- \seq_rand_item:N
 [161](#), [17914](#), [17914](#), [17919](#)
- \seq_remove_all:Nn . [157](#), [162](#), [168](#),
 [169](#), [17592](#), [17592](#), [17616](#), [17617](#), [19438](#)
- \seq_remove_duplicates:N [162](#), [168](#),
 [169](#), [11642](#), [17576](#), [17576](#), [17590](#), [40695](#)
- \seq_reverse:N
 [163](#), [865](#), [17686](#), [17686](#), [17702](#)
- \seq_set_eq:NN [156](#), [169](#),
 [3310](#), [17390](#), [17401](#), [17401](#), [17402](#),
 [17403](#), [17404](#), [17520](#), [17577](#), [17715](#),
 [40600](#), [40710](#), [40747](#), [42109](#), [42183](#)
- \seq_set_filter:NNn
 [158](#), [875](#), [6200](#), [17486](#), [17486](#)
- \seq_set_from_clist:NN
 [157](#), [17409](#), [17409](#), [17429](#), [17430](#), [19437](#)
- \seq_set_from_clist:Nn . [157](#), [191](#),
 [858](#), [11638](#), [11652](#), [17409](#), [17414](#), [17431](#)
- \seq_set_item:Nnn
 [162](#), [17621](#), [17621](#), [17625](#), [17627](#), [17631](#)
- \seq_set_item:NnnTF [162](#), [17621](#)
- \seq_set_map:NNn ... [165](#), [18078](#), [18078](#)
- \seq_set_map_e:NNn [166](#),
 [876](#), [6213](#), [18068](#), [18068](#), [41538](#), [41539](#)
- \seq_set_map_x:NNn [41538](#), [41539](#)
- \seq_set_regex_extract_all:NNn ..
 [158](#), [17496](#), [17505](#), [17507](#)
- \seq_set_regex_extract_all:Nnn ..
 [158](#), [17496](#), [17502](#), [17504](#)
- \seq_set_regex_extract_once:NNn .
 [158](#), [17496](#), [17499](#), [17501](#)
- \seq_set_regex_extract_once:Nnn .
 [158](#), [17496](#), [17496](#), [17498](#)
- \seq_set_regex_split:NNn
 [159](#), [17496](#), [17511](#), [17513](#)
- \seq_set_regex_split:Nnn
 [159](#), [17496](#), [17508](#), [17510](#)
- \seq_set_split:Nnn
 ... [157](#), [6199](#), [6212](#), [9231](#), [17441](#),
 [17441](#), [17480](#), [17481](#), [19963](#), [19966](#)

- \seq_set_split_keep_spaces:Nnn ..
..... 157, 17441, 17445, 17484
- \seq_show:N ..
.. 170, 641, 752, 18167, 18167, 18168
- \seq_shuffle:N 163, 17714, 17715, 17751
- \seq_sort:Nn ..
.... 47, 163, 3309, 3309, 3311, 17704
- \seq_use:Nn ..
.... 167, 6216, 18112, 18146, 18148
- \seq_use:Nnnn ..
.... 166, 18112, 18112, 18134, 18147
- \g_tmpa_seq .. 170, 18192
- \l_tmpa_seq .. 170, 18192
- \g_tmpb_seq .. 170, 18192
- \l_tmpb_seq .. 170, 18192
- seq internal commands:
- __seq_count:w ..
 876, 18088, 18093, 18106, 18109, 18110
- __seq_count_end:w .. 876,
 18088, 18095, 18096, 18097, 18098,
 18099, 18100, 18101, 18102, 18110
- __seq_get_left:wnw ..
 17792, 17796, 17800
- __seq_get_right_end:NnN ..
 17817, 17825, 17833
- __seq_get_right_loop:nw ..
 869, 17817, 17822, 17828, 17831
- __seq_if_in: ... 17753, 17762, 17770
- __seq_int_eval:w ..
 17620, 17620, 17667, 17677
- __seq_item:n ..
 839, 856, 860, 862, 867–869, 871–
 873, 875–877, 17374, 17374, 17549,
 17557, 17567, 17569, 17574, 17635,
 17672, 17675, 17692, 17693, 17695,
 17700, 17728, 17758, 17797, 17800,
 17810, 17825, 17828, 17841, 17842,
 17853, 17897, 17906, 17931, 17936,
 17937, 17938, 17939, 17951, 17956,
 17962, 17966, 17982, 17988, 17989,
 17990, 17991, 18035, 18037, 18074,
 18084, 18095, 18096, 18097, 18098,
 18099, 18100, 18101, 18102, 18107,
 18108, 18123, 18138, 18141, 18144
- __seq_item:nN .. 17890, 17895, 17900
- __seq_item:nwn ..
 17890, 17894, 17906, 17911
- __seq_item:wNn .. 17890, 17891, 17892
- __seq_map_function:Nw ..
 872, 17924, 17927, 17935, 17945
- __seq_map_indexed:NN ..
 18014, 18022, 18027
- __seq_map_indexed:nNN .. 18012
- __seq_map_indexed:Nw ..
 874, 18012, 18029, 18037, 18041
- __seq_map_pairwise_function:Nnnwnn ..
 18045, 18056, 18060, 18065
- __seq_map_pairwise_function:wNN ..
 18045, 18046, 18047
- __seq_map_pairwise_function:wNw ..
 18045, 18049, 18054
- __seq_map_tokens:nw ..
 17977, 17980, 17987, 17997
- __seq_pop:NNNN .. 17774,
 17774, 17804, 17806, 17836, 17838
- __seq_pop_item_def: ..
 856, 17494, 17614, 17730, 17948,
 17964, 17974, 18007, 18076, 18086
- __seq_pop_left:NNN .. 17803,
 17804, 17806, 17807, 17872, 17875
- __seq_pop_left:wnwNNN ..
 17803, 17808, 17809
- __seq_pop_right:NNN .. 863, 17835,
 17836, 17838, 17839, 17878, 17881
- __seq_pop_right_loop:nn ..
 17835, 17846, 17855, 17858
- __seq_pop_TF:NNNN ..
 870, 17774, 17782, 17863,
 17865, 17872, 17875, 17878, 17881
- __seq_push_item_def: ..
 17727, 17948, 17950, 17955, 17958
- __seq_push_item_def:n ..
 856, 17492, 17598, 17948, 17948,
 17953, 17972, 18001, 18074, 18084
- __seq_put_left_aux:w ..
 861, 17545, 17550, 17558, 17561
- __seq_remove_all_aux:Nnn ..
 17592, 17593, 17595, 17596
- __seq_remove_duplicates:NN ..
 17576, 17577, 17579, 17580
- __seq_reverse:NN ..
 17686, 17687, 17689, 17690
- __seq_reverse_item:nw .. 865
- __seq_reverse_item:nwn ..
 17686, 17693, 17697
- __seq_sep: ..
 18011, 18011, 18033, 18037, 18043
- __seq_set_filter:NNNn ..
 17486, 17487, 17489, 17490
- __seq_set_item:NnnNN .. 17621,
 17622, 17624, 17628, 17630, 17633
- __seq_set_item:nnNNNN ..
 17621, 17636, 17639
- __seq_set_item:nNnnNNNN ..
 864, 17621, 17642, 17647, 17661
- __seq_set_item:wn ..
 17621, 17666, 17672, 17676

- __seq_set_item_end:w 864, 17621, 17674, 17679
- __seq_set_item_false:nnNNNN ... 864, 17621, 17650, 17652
- __seq_set_map:NNnn 18078, 18079, 18081, 18082
- __seq_set_map_e:NNnn 18068, 18069, 18071, 18072
- __seq_set_split:NNnn 17441
- __seq_set_split:NNnn 17442, 17444, 17446, 17448, 17449
- __seq_set_split:Nw 859, 17441, 17459, 17466, 17472
- __seq_set_split:w 859, 17441, 17474, 17478
- __seq_set_split_end: 859, 17441, 17461, 17465, 17472, 17476, 17478
- __seq_show:NN 18167, 18167, 18169, 18171
- __seq_show_validate:nn 18167, 18176, 18186, 18190
- __seq_shuffle:NN 17714, 17715, 17716, 17717
- __seq_shuffle_item:n 17714, 17728, 17742
- __seq_tmp:w . 17381, 17381, 17515, 17528, 17529, 17530, 17531, 17532, 17533, 17692, 17695, 17841, 17853
- \g_seq_tmp_seq 17714
- \l_seq_tmp_seq 17520, 17522, 17523, 17575, 17582, 17585, 17586, 17588
- \l_seq_tmpa_int 17726, 17732, 17744, 17746, 17747
- \l_seq_tmpa_tl 859, 864, 17379, 17453, 17457, 17463, 17468, 17470, 17607, 17612, 17635, 17682, 17757, 17761
- \l_seq_tmpb_int . 17745, 17748, 17749
- \l_seq_tmpb_tl 17379, 17603, 17607, 17760, 17761
- __seq_use:NNnnNnn 18112, 18119, 18120, 18135
- __seq_use:nwnn . 18112, 18125, 18144
- __seq_use:nwwwnwnn 18112, 18124, 18136, 18137
- __seq_use_setup:w 18112, 18123, 18136
- __seq_wrap_item:n 859, 860, 17412, 17417, 17422, 17427, 17438, 17454, 17479, 17492, 17574, 17574, 17610, 18189
- \setbox 395
- \setfontid 914
- \setlanguage 396
- \setrandomseed 960
- \sfcode 397
- \sffamily 38409
- \shapemode 915
- \shbscode 681
- \shellescape 777
- \Shipout 1249
- \shipout 398, 1236, 1237
- \ShortText 37, 75
- \show 399
- \showbox 400
- \showboxbreadth 401
- \showboxdepth 402
- \showgroups 522
- \showifs 523
- \showlists 403
- \showmode 1185
- \showstream 1217
- \showthe 404
- \showtokens 524
- sign 283
- sin 283
- sind 284
- \sjis 1186
- \skewchar 405
- \skip 406, 20333
- skip commands:
 - \c_max_skip 241, 22143
 - \skip_add:Nn 239, 22093, 22093, 22097, 42187, 42562
 - \skip_const:Nn 239, 1004, 22063, 22063, 22068, 22143, 22144, 42311, 42566
 - \skip_eval:n ... 240, 22066, 22107, 22123, 22123, 22138, 22142, 42607
 - \skip_gadd:Nn 239, 22093, 22095, 22098, 42269, 42563
 - .skip_gset:N 250, 23084
 - \skip_gset:Nn 239, 1000, 22083, 22085, 22088, 42267, 42561
 - \skip_gset_eq:NN 239, 22089, 22091, 22092, 42268
 - \skip_gsub:Nn 239, 22093, 22101, 22104, 42270, 42565
 - \skip_gzero:N 239, 22069, 22070, 22072, 22076, 42266
 - \skip_gzero_new:N 239, 22073, 22075, 22078
 - \skip_horizontal:N 241, 22127, 22127, 22133
 - \skip_horizontal:n 241, 22127, 22128, 42608
 - \skip_if_eq:nn 22105
 - \skip_if_eq:nnTF 240, 22105
 - \skip_if_eq_p:nn 240, 22105

- \skip_if_exist:N 22079, 22081
- \skip_if_exist:NTF
..... 239, 22074, 22076, 22079
- \skip_if_exist_p:N 239, 22079
- \skip_if_finite:n 22114, 42730, 42735
- \skip_if_finite:nTF 240, 22112
- \skip_if_finite_p:n 240, 22112
- \skip_log:N .. 241, 22139, 22139, 22140
- \skip_log:n 241, 22139, 22141
- \skip_new:N 238, 239,
22057, 22057, 22062, 22065, 22074,
22076, 22145, 22146, 22147, 22148
- .skip_set:N 250, 23084
- \skip_set:Nn
239, 22083, 22083, 22087, 42185, 42560
- \skip_set_eq:NN
..... 239, 22089, 22089, 22090, 42186
- \skip_show:N . 240, 22135, 22135, 22136
- \skip_show:n . 240, 1003, 22137, 22137
- \skip_sub:Nn
239, 22093, 22099, 22103, 42188, 42564
- \skip_use:N 240, 22117,
22124, 22125, 22125, 22126, 42733
- \skip_vertical:N
..... 241, 22127, 22130, 22134
- \skip_vertical:n
..... 241, 22127, 22131, 42609
- \skip_zero:N 239, 242,
985, 22069, 22069, 22071, 22074, 42184
- \skip_zero_new:N
..... 239, 22073, 22073, 22077
- \g_tmpa_skip 241, 22145
- \l_tmpa_skip 241, 22145
- \g_tmpb_skip 241, 22145
- \l_tmpb_skip 241, 22145
- \c_zero_skip
..... 241, 985, 21598, 21600, 22143
- skip internal commands:
- _skip_if_finite:wwNw
..... 22112, 22116, 22120, 42732
- _skip_sep: 22111, 22111,
22117, 22118, 22120, 42736, 42737
- _skip_tmp:w
..... 22112, 22122, 42728, 42740
- \skipdef 407
- sort commands:
- \sort_return_same:
..... 46, 47, 454, 3393, 3393
- \sort_return_swapped:
..... 46, 47, 454, 3393, 3403
- sort internal commands:
- _sort:nnNnn 456, 457
- _l__sort_A_int .. 453, 3191, 3198,
3205, 3208, 3217, 3357, 3362, 3365,
3385, 3417, 3424, 3439, 3441, 3442
- _l__sort_B_int 453, 3191,
3362, 3366, 3374, 3376, 3377, 3429,
3430, 3439, 3440, 3449, 3450, 3452
- _l__sort_begin_int
..... 448, 453, 3189, 3354, 3442, 3452
- _l__sort_block_int 448, 452,
3188, 3200, 3205, 3209, 3212, 3217,
3218, 3283, 3345, 3348, 3355, 3358
- _l__sort_C_int 453, 3191,
3363, 3367, 3374, 3375, 3386, 3418,
3425, 3429, 3431, 3432, 3449, 3451
- ___sort_compare:nn 450, 454, 3282, 3384
- ___sort_compute_range:
..... 447-449, 3222,
3222, 3230, 3238, 3246, 3259, 3270
- ___sort_copy_block:
..... 452, 3364, 3372, 3372, 3380
- ___sort_disable_toksdef:
..... 3269, 3549, 3549
- ___sort_disabled_toksdef:n
..... 3549, 3550, 3551
- _l__sort_end_int 448,
452, 453, 3189, 3346, 3354, 3355,
3356, 3357, 3358, 3359, 3360, 3377
- ___sort_error: 3543, 3543, 3555, 3573
- ___sort_i:nnnnNn 457
- _l__sort_length_int
..... 447, 448, 3183, 3280, 3345
- ___sort_level:
450, 460, 3284, 3343, 3343, 3349, 3547
- ___sort_loop:wNn 456, 457
- ___sort_main:NNNn
..... 451, 3267, 3267, 3293, 3330
- _l__sort_max_int
..... 447, 448, 3183, 3202, 3274
- _c__sort_max_length_int 3222
- ___sort_merge_blocks:
..... 3347, 3352, 3352, 3369, 3546
- ___sort_merge_blocks_aux:
452, 3368, 3382, 3382, 3435, 3445, 3545
- ___sort_merge_blocks_end:
..... 455, 3443, 3447, 3447, 3455
- _l__sort_min_int
.. 447, 448, 450, 451, 3183, 3199,
3207, 3224, 3240, 3248, 3261, 3271,
3281, 3295, 3333, 3346, 3571, 3572
- ___sort_quick_cleanup:w
..... 3457, 3478, 3481
- ___sort_quick_end:nnTFNn
..... 458, 459, 3477,
3517, 3517, 3523, 3530, 3535, 3538

- __sort_quick_only_i:NnnnnNn . . .
 [3482](#), [3485](#), [3489](#), [3492](#)
- __sort_quick_only_i_end:nnwnw .
 [3493](#), [3517](#), [3520](#)
- __sort_quick_only_ii:NnnnnNn . . .
 [3482](#), [3484](#), [3496](#), [3498](#)
- __sort_quick_only_ii_end:nnwnw
 [3500](#), [3517](#), [3527](#)
- __sort_quick_prepare:Nnnn
 [3457](#), [3463](#), [3470](#), [3473](#)
- __sort_quick_prepare_end:NNNnw .
 [3457](#), [3465](#), [3475](#)
- __sort_quick_single_end:nnwnw .
 [3486](#), [3517](#), [3518](#)
- __sort_quick_split:NnNn
 [457](#), [458](#), [3477](#),
 [3482](#), [3482](#), [3522](#), [3529](#), [3535](#), [3537](#)
- __sort_quick_split_end:nnwnw . .
 [3507](#), [3514](#), [3517](#), [3533](#)
- __sort_quick_split_i:NnnnnNn . . .
 . . . [456](#), [3482](#), [3499](#), [3503](#), [3506](#), [3513](#)
- __sort_quick_split_ii:NnnnnNn . .
 [3482](#), [3491](#), [3505](#), [3510](#), [3512](#)
- __sort_redefine_compute_range: .
 [3222](#), [3229](#), [3234](#), [3254](#)
- __sort_return_mark:w
 [454](#), [3388](#), [3389](#),
 [3393](#), [3394](#), [3399](#), [3404](#), [3409](#), [3413](#)
- __sort_return_none_error:
 [454](#), [3391](#), [3393](#), [3414](#), [3419](#), [3427](#), [3437](#)
- __sort_return_same:w
 [454](#), [3401](#), [3419](#), [3427](#), [3427](#)
- __sort_return_swapped:w
 [3411](#), [3437](#), [3437](#)
- __sort_return_two_error:
 [454](#), [3393](#), [3398](#), [3408](#), [3421](#)
- __sort_sep:
 [3180](#), [3180](#), [3295](#), [3301](#), [3306](#)
- __sort_seq:NNNNn
 [450](#), [3309](#), [3310](#), [3313](#), [3317](#), [3323](#), [3327](#)
- __sort_shrink_range: [448](#),
 [449](#), [3196](#), [3196](#), [3226](#), [3242](#), [3250](#), [3263](#)
- __sort_shrink_range_loop:
 [3196](#), [3201](#), [3215](#), [3219](#)
- __sort_tl:NnN
 [450](#), [3286](#), [3286](#), [3288](#), [3290](#)
- __sort_tl_toks:w
 [451](#), [3286](#), [3295](#), [3301](#), [3305](#)
- \g__sort_tmp_seq
 [450](#), [451](#), [3181](#), [3331](#), [3335](#), [3339](#), [3340](#)
- \g__sort_tmp_tl [3181](#), [3294](#), [3297](#), [3298](#)
- __sort_too_long_error:NNw
 [3275](#), [3566](#), [3566](#)
- \l__sort_top_int
 . . . [447](#), [450](#), [451](#), [453](#), [3183](#), [3271](#),
 [3274](#), [3277](#), [3278](#), [3281](#), [3303](#), [3333](#),
 [3356](#), [3359](#), [3360](#), [3363](#), [3432](#), [3572](#)
- \l__sort_true_max_int
 [447](#), [448](#), [3183](#), [3199](#),
 [3212](#), [3225](#), [3241](#), [3249](#), [3262](#), [3571](#)
- sp [287](#)
- \spacefactor [408](#)
- \spaceskip [409](#)
- \span [410](#)
- \special [411](#)
- \splitbotmark [412](#)
- \splitbotmarks [525](#)
- \splitdiscards [526](#)
- \splitfirstmark [413](#)
- \splitfirstmarks [527](#)
- \splitmaxdepth [414](#)
- \splittopskip [415](#)
- sqrt [285](#)
- \SS [33346](#), [34887](#), [35785](#)
- \ss [33346](#), [34887](#), [35781](#)
- \stbscode [682](#)
- \stockheight [41139](#), [41147](#), [41151](#), [41155](#)
- \stockwidth [41140](#), [41148](#), [41151](#), [41156](#)
- str commands:
- \c_ampersand_str [145](#), [14417](#)
- \c_atsign_str [145](#), [14417](#)
- \c_backslash_str
 [145](#), [4674](#), [5277](#), [14417](#),
 [15122](#), [15124](#), [15147](#), [15176](#), [15178](#),
 [15210](#), [15219](#), [15223](#), [42046](#), [42056](#)
- \c_circumflex_str [145](#), [14417](#)
- \c_colon_str [145](#), [14417](#), [20244](#),
 [20449](#), [20455](#), [22612](#), [22618](#), [39433](#),
 [39438](#), [40547](#), [40556](#), [40774](#), [40784](#)
- \c_dollar_str [145](#), [14417](#)
- \c_empty_str [145](#), [14430](#)
- \c_hash_str [145](#), [14417](#),
 [15090](#), [15193](#), [15767](#), [15768](#), [15771](#),
 [15774](#), [32206](#), [32532](#), [32561](#), [32565](#),
 [41944](#), [42514](#), [42516](#), [42576](#), [42624](#),
 [42629](#), [42659](#), [42663](#), [42665](#), [42683](#)
- \c_left_brace_str [145](#), [507](#), [4737](#),
 [5152](#), [5156](#), [5176](#), [5189](#), [5213](#), [5685](#),
 [5696](#), [5700](#), [5779](#), [5803](#), [6974](#), [14417](#)
- \c_percent_str
 [145](#), [14417](#), [15092](#), [15246](#), [40559](#), [40787](#)
- \c_right_brace_str
 [145](#), [4773](#), [5162](#), [5182](#), [5195](#),
 [5703](#), [5707](#), [5800](#), [6971](#), [14417](#), [23714](#)
- \str_case:Nn [137](#), [13862](#), [13887](#)

- \str_case:nn [137](#),
[5417](#), [8808](#), [10182](#), [13862](#), [13862](#),
[13884](#), [13885](#), [13887](#), [39951](#), [39989](#)
- \str_case:NnTF
..... [137](#), [13862](#), [13888](#), [13889](#), [13890](#)
- \str_case:nnTF [137](#), [612](#), [890](#),
[989](#), [1397](#), [6027](#), [8884](#), [8919](#), [9285](#),
[9998](#), [13862](#), [13867](#), [13872](#), [13877](#),
[13888](#), [13889](#), [13890](#), [16855](#), [16868](#),
[16911](#), [16926](#), [16959](#), [16993](#), [17019](#),
[22826](#), [22871](#), [26517](#), [30738](#), [30752](#),
[35133](#), [35347](#), [35365](#), [35391](#), [35457](#)
- \str_case_e:nn
..... [137](#), [13862](#), [13897](#), [13919](#), [13920](#)
- \str_case_e:nnTF [137](#), [5160](#),
[13862](#), [13902](#), [13907](#), [13912](#), [15145](#)
- \str_casefold:n [143](#), [144](#),
[304](#), [14281](#), [14281](#), [14284](#), [25325](#),
[36084](#), [36338](#), [39620](#), [40620](#), [41520](#),
[41521](#), [41522](#), [41523](#), [41524](#), [41525](#),
[41526](#), [41527](#), [41611](#), [41612](#), [41619](#),
[41620](#), [41627](#), [41628](#), [41637](#), [41638](#)
- \c_str_cctab [298](#), [1310](#), [31806](#)
- \str_clear:N
..... [135](#), [13691](#), [22591](#), [22762](#),
[23276](#), [23277](#), [40735](#), [40742](#), [42189](#)
- \str_clear_new:N [135](#), [13691](#)
- \str_compare:nNn [13817](#), [13824](#)
- \str_compare:nNnTF [138](#), [13817](#)
- \str_compare_p:nNn [138](#), [13817](#)
- \str_concat:NNN
..... [135](#), [13691](#), [13714](#), [13716](#), [42129](#)
- \str_const:Nn
..... [135](#), [8734](#), [8756](#), [8774](#), [8806](#),
[8875](#), [9105](#), [9247](#), [11808](#), [11815](#),
[11819](#), [11823](#), [13718](#), [13722](#), [13749](#),
[14417](#), [14418](#), [14419](#), [14420](#), [14421](#),
[14422](#), [14423](#), [14424](#), [14425](#), [14426](#),
[14427](#), [14428](#), [14429](#), [15186](#), [15187](#),
[15209](#), [22450](#), [22451](#), [22452](#), [22453](#),
[22454](#), [22455](#), [22456](#), [32149](#), [42312](#)
- \str_convert_pdfname:n
..... [148](#), [15745](#), [15745](#), [39717](#)
- \str_count:N [140](#), [4084](#), [9426](#), [9427](#),
[9600](#), [9601](#), [10688](#), [10766](#), [14211](#),
[14211](#), [14212](#), [31342](#), [31347](#), [31423](#)
- \str_count:n
..... [140](#), [4078](#), [14211](#), [14211](#), [14213](#), [14221](#)
- \str_count_ignore_spaces:n
..... [140](#), [772](#), [3657](#), [14211](#), [14227](#)
- \str_count_spaces:N
..... [140](#), [14191](#), [14191](#), [14193](#)
- \str_count_spaces:n
..... [140](#), [772](#), [14191](#), [14192](#), [14194](#), [14217](#)
- \str_declare_eight_bit_encoding:nnn
..... [41528](#), [41529](#)
- \str_fold_case:n . [41512](#), [41521](#), [41523](#)
- \str_foldcase:n . [41524](#), [41525](#), [41527](#)
- \str_gclear:N [135](#), [13691](#), [42271](#)
- \str_gclear_new:N [135](#), [13691](#)
- \str_gconcat:NNN
..... [135](#), [13691](#), [13715](#), [13717](#), [42130](#)
- \str_gput_left:Nn
..... [136](#), [13718](#), [13732](#), [13751](#), [42273](#)
- \str_gput_right:Nn
..... [136](#), [13718](#), [13742](#), [13753](#), [42274](#)
- \str_gremove_all:Nn
..... [142](#), [13800](#), [13802](#), [13805](#)
- \str_gremove_once:Nn
..... [142](#), [13794](#), [13796](#), [13799](#)
- \str_greplace_all:Nnn
..... [142](#), [13754](#), [13760](#), [13765](#), [13803](#)
- \str_greplace_once:Nnn
..... [142](#), [13754](#), [13756](#), [13763](#), [13797](#)
- .str_gset:N [250](#), [23092](#)
- \str_gset:Nn
..... [135](#), [11554](#), [11555](#), [11556](#),
[13718](#), [13720](#), [13748](#), [31333](#), [31337](#)
- \str_gset_convert:Nnnn
..... [148](#), [14626](#), [14628](#), [14640](#)
- \str_gset_convert:NnnnTF . [148](#), [14626](#)
- .str_gset_e:N [250](#), [23092](#)
- \str_gset_eq:NN
. [135](#), [11541](#), [11542](#), [11543](#), [13691](#),
[13711](#), [13713](#), [31417](#), [42112](#), [42272](#)
- .str_gset_x:N [41438](#)
- \str_head:N
..... [140](#), [773](#), [14251](#), [14251](#), [14252](#)
- \str_head:n [140](#), [742](#),
[773](#), [13176](#), [13223](#), [14251](#), [14251](#), [14253](#)
- \str_head_ignore_spaces:n
..... [140](#), [14251](#), [14261](#)
- \str_if_empty:N [13810](#), [13812](#)
- \str_if_empty:n [13814](#)
- \str_if_empty:NnTF [136](#), [13806](#),
[22546](#), [22569](#), [23283](#), [23552](#), [40613](#)
- \str_if_empty:nTF
..... [136](#), [13806](#), [36320](#), [36329](#), [36352](#)
- \str_if_empty_p:N [136](#), [13806](#)
- \str_if_empty_p:n [136](#), [13806](#)
- \str_if_eq:NN [13841](#), [13846](#)
- \str_if_eq:nn [976](#), [13829](#), [13834](#), [13836](#)
- \str_if_eq:NNTF [136](#), [762](#), [13841](#)
- \str_if_eq:nnTF
..... [103](#), [115](#), [136](#), [137](#), [219](#), [225](#),
[226](#), [863](#), [943](#), [5062](#), [8185](#), [8329](#),
[8792](#), [8868](#), [8899](#), [9076](#), [9758](#), [9801](#),
[10098](#), [10101](#), [10166](#), [11613](#), [11676](#),

- 11691, 12613, [13829](#), 13893, 13923,
15530, 15533, 15688, 15691, 17600,
20247, 20304, 21118, 21372, 22107,
22660, 25195, 25268, 26541, 26552,
26561, 30727, 32206, 32543, 32561,
32563, 33116, 33151, 33207, 33575,
33602, 33703, 35176, 35222, 35297,
35352, 35681, 36282, 36305, 38470,
38474, 38855, 39228, 39387, 39422,
39467, 39526, 39819, 39829, 40558,
40566, 40603, 40713, 40720, 40786
- \str_if_eq_p:NN [136](#), [13841](#)
- \str_if_eq_p:nn
.... [136](#), 8749, 8779, 8780, 8907,
8908, 9255, 9257, 11827, [13829](#),
32680, 32948, 32966, 33212, 36292,
36293, 37503, 38406, 39222, 40875
- \str_if_exist:N
..... [13806](#), 13808, 31327, 31329
- \str_if_exist:NTF
..... [135](#), 8866, 8935, [13806](#)
- \str_if_exist_p:N [135](#), [13806](#)
- \str_if_in:Nn [13848](#), [13854](#)
- \str_if_in:nn [13856](#)
- \str_if_in:NnTF [136](#), [13848](#)
- \str_if_in:nnTF [137](#), 3112, [13848](#), 31790
- \str_ignore_spaces:n [14232](#)
- \str_item:Nn
.... [141](#), [14053](#), 14053, 14054, 31449
- \str_item:nn
.. [141](#), 768, 772, [14053](#), 14053, 14055
- \str_item_ignore_spaces:nn
..... [141](#), 768, [14053](#), 14063
- \str_log:N ... [144](#), [14435](#), 14443, 14448
- \str_log:n [144](#), [14435](#), 14442
- \str_lower_case:n [41512](#), 41513, 41515
- \str_lowercase:n . [143](#), 304, [14281](#),
14282, 14285, 41512, 41513, 41514,
41515, 41613, 41614, 41629, 41630
- \str_map_break: [139](#), [13929](#),
13935, 13944, 13961, 13969, 13981,
13987, 13993, 13994, 13996, 14003
- \str_map_break:n
..... [139](#), 3116, 5926, [13929](#), 13995
- \str_map_function:NN
..... [138](#), 766, [13929](#), 13937, 13948
- \str_map_function:nN [138](#),
765, 5919, [13929](#), 13929, 13938, 15748
- \str_map_inline:Nn
..... [138](#), [139](#), [13929](#), 13964, 13966
- \str_map_inline:nn
.. [138](#), 3110, 6672, [13929](#), 13949, 13965
- \str_map_tokens:Nn
..... [138](#), [13997](#), 14005, 14006
- \str_map_tokens:nn
..... [138](#), [13997](#), 13997, 14005
- \str_map_variable:NNn
..... [139](#), [13929](#), 13983, 13992
- \str_map_variable:nNn
..... [139](#), [13929](#), 13973, 13984
- \str_md5five_hash:n
.... [144](#), [14414](#), 14414, 14415, 14416
- \str_new:N [135](#), 9307,
9308, 11004, 11005, 11006, 11035,
11036, 11037, 11844, [13691](#), 14431,
14432, 14433, 14434, 22458, 22460,
22464, 22466, 22469, 22471, 22474,
40585, 40586, 40588, 40589, 40590
- \str_put_left:Nn
..... [136](#), 759, [13718](#), 13727, 13750, 42191
- \str_put_right:Nn
..... [136](#), 759, [13718](#), 13737, 13752, 42192
- \str_range:Nnn
..... [141](#), [14114](#), 14114, 14115, 31354, 31356
- \str_range:nnn [103](#), [141](#), 772,
4081, 6029, [14114](#), 14114, 14116, 16975
- \str_range_ignore_spaces:nnn ...
..... [141](#), [14114](#), 14124
- \str_remove_all:Nn
..... [142](#), [13800](#), 13800, 13804
- \str_remove_once:Nn
..... [142](#), [13794](#), 13794, 13798
- \str_replace_all:Nnn
.... [142](#), [13754](#), 13758, 13764, 13801
- \str_replace_once:Nnn
.... [142](#), [13754](#), 13754, 13762, 13795
- .str_set:N [250](#), [23092](#)
- \str_set:Nn
..... [135](#), [142](#), 250, 1019, 9389,
9390, 9588, 9589, 11626, 11627,
11628, [13718](#), 13718, 13747, 13988,
22514, 22696, 23178, 23180, 23280,
23291, 23440, 31331, 31335, 40638
- \str_set_convert:Nnnn [148](#),
[149](#), 784, 795, [14626](#), 14626, 14631
- \str_set_convert:NnnnTF
..... [148](#), 784, [14626](#)
- .str_set_e:N [250](#), [23092](#)
- \str_set_eq:NN [135](#), [13691](#),
13710, 13712, 31413, 42111, 42190
- .str_set_x:N [41438](#)
- \str_show:N .. [144](#), [14435](#), 14436, 14441
- \str_show:n [144](#), [14435](#), 14435
- \str_tail:N .. [140](#), [14266](#), 14266, 14267
- \str_tail:n [140](#),
[469](#), [14266](#), 14266, 14268, 32732, 40619
- \str_tail_ignore_spaces:n
..... [140](#), [14266](#), 14275

- \str_titlecase:n 41617, 41618
- \str_upper_case:n [41512](#), 41517, 41519
- \str_uppercase:n . [143](#), [304](#), [14281](#),
14283, 14286, 41516, 41517, 41518,
41519, 41615, 41616, 41635, 41636
- \str_use:N [140](#), [13691](#)
- \c_tilde_str [145](#), [14417](#)
- \g_tmpa_str [145](#), [14431](#)
- \l_tmpa_str [142](#), [145](#), [14431](#)
- \g_tmpb_str [145](#), [14431](#)
- \l_tmpb_str [145](#), [14431](#)
- \c_underscore_str .. [145](#), [14417](#), 31047
- \c_zero_str
[145](#), [14417](#), 31307, 31313, 31413, 31417
- str internal commands:
- \g_str_alias_prop . [787](#), [14459](#), 14699
- \c__str_byte_1_tl [14540](#)
- \c__str_byte_0_tl [14540](#)
- \c__str_byte_1_tl [14540](#)
- \c__str_byte_255_tl [14540](#)
- \c__str_byte_⟨number⟩_tl [782](#)
- \l__str_byte_flag
..... [789](#), [14487](#), 14767, 14781,
[14784](#), 15049, 15058, 15102, 15117
- __str_case:nnTF [13862](#),
13865, 13870, 13875, 13880, 13882
- __str_case:nw
..... [13862](#), 13883, 13891, 13895
- __str_case_e:nnTF [13862](#),
13900, 13905, 13910, 13915, 13917
- __str_case_e:nw
..... [13862](#), 13918, 13921, 13925
- __str_case_end:nw
..... [13862](#), 13894, 13924, 13927
- __str_change_case:nn
.. [14281](#), [14281](#), [14282](#), [14283](#), [14287](#)
- __str_change_case_aux:nn
..... [14281](#), [14289](#), [14292](#)
- __str_change_case_char:nN
..... [14281](#), [14306](#), [14315](#)
- __str_change_case_char:nnn
..... [14281](#), [14326](#), [14355](#),
[14358](#), [14364](#), [14373](#), [14386](#), [14395](#)
- __str_change_case_char:nnnnn ...
..... [14281](#), [14398](#), [14400](#)
- __str_change_case_char_aux:nnn .
..... [14281](#), [14390](#), [14396](#)
- __str_change_case_char_auxi:nN .
..... [14281](#), [14333](#), [14337](#), [14343](#)
- __str_change_case_char_auxii:nN
..... [14281](#), [14336](#), [14340](#), [14354](#)
- __str_change_case_codepoint:nN .
..... [14281](#), [14319](#), [14325](#), [14328](#)
- __str_change_case_codepoint:nNN
..... [14281](#), [14346](#), [14356](#)
- __str_change_case_codepoint:nNNN
..... [14281](#), [14349](#), [14362](#)
- __str_change_case_codepoint:nNNNN
..... [14281](#), [14371](#)
- __str_change_case_codepoint:nNNNNN
..... [14350](#)
- __str_change_case_end:nw [14281](#)
- __str_change_case_end:wn
..... [14300](#), [14318](#)
- __str_change_case_loop:nw
.. [14281](#), [14294](#), [14302](#), [14313](#), [14393](#)
- __str_change_case_output:nw ...
.. [14281](#), [14297](#), [14299](#), [14312](#), [14388](#)
- __str_change_case_result:n
.. [14281](#), [14295](#), [14297](#), [14298](#), [14300](#)
- __str_change_case_space:n
..... [14281](#), [14305](#), [14310](#)
- __str_collect_delimit_by_q_-
stop:w [14142](#), [14165](#), [14165](#)
- __str_collect_end:nnnnnnnnnw ...
..... [771](#), [14165](#), [14184](#), [14189](#)
- __str_collect_end:wn
..... [14165](#), [14172](#), [14182](#)
- __str_collect_loop:wn
..... [14165](#), [14166](#), [14167](#), [14178](#)
- __str_collect_loop:wnNNNNNNN ...
..... [14165](#), [14170](#), [14176](#)
- __str_convert:nnn
[786](#), [787](#), [14671](#), [14672](#), [14686](#), [14686](#)
- __str_convert:nnnn
..... [787](#), [14686](#), [14690](#), [14695](#)
- __str_convert:NNnNN
..... [14668](#), [14673](#), [14676](#)
- __str_convert:nNNnnn ... [14626](#),
[14627](#), [14629](#), [14634](#), [14643](#), [14648](#)
- __str_convert:wwwnn
.... [786](#), [14653](#), [14658](#), [14668](#), [14668](#)
- __str_convert_decode_:
..... [14657](#), [14791](#), [14791](#)
- __str_convert_decode_clist: ...
..... [14831](#), [14831](#)
- __str_convert_decode_eight_-
bit:n [14852](#), [14896](#), [14896](#)
- __str_convert_decode_utf16: . [15519](#)
- __str_convert_decode_utf16be: [15519](#)
- __str_convert_decode_utf16le: [15519](#)
- __str_convert_decode_utf32: . [15677](#)
- __str_convert_decode_utf32be: [15677](#)
- __str_convert_decode_utf32le: [15677](#)
- __str_convert_decode_utf8: .. [15338](#)
- __str_convert_encode:
..... [14662](#), [14795](#), [14801](#), [14807](#)

```

\\_str_convert_encode_clist: ...
    ..... 14842, 14842
\\_str_convert_encode_eight_-
    bit:n ..... 14854, 14923, 14924
\\_str_convert_encode_utf16: . 15434
\\_str_convert_encode_utf16be: 15434
\\_str_convert_encode_utf16le: 15434
\\_str_convert_encode_utf32: . 15617
\\_str_convert_encode_utf32be: 15617
\\_str_convert_encode_utf32le: 15617
\\_str_convert_encode_utf8: .. 15262
\\_str_convert_escape_: .....
    ..... 14789, 14789, 14790
\\_str_convert_escape_bytes: ...
    ..... 14789, 14790
\\_str_convert_escape_hex: .....
    ..... 15182, 15182
\\_str_convert_escape_name: .....
    ..... 802, 15186, 15188
\\_str_convert_escape_string: ...
    ..... 15209, 15211
\\_str_convert_escape_url: .....
    ..... 15241, 15241
\\_str_convert_gmap:N .....
    ..... 14584, 14584, 14792,
    14904, 15183, 15189, 15212, 15242
\\_str_convert_gmap_internal:N ..
    ..... 14600,
    14600, 14802, 14810, 14844, 14933,
    15263, 15447, 15619, 15623, 15625
\\_str_convert_gmap_internal_-
    loop:Nw ..... 14600
\\_str_convert_gmap_internal_-
    loop:Nww ..... 14604, 14610, 14614
\\_str_convert_gmap_loop:NN .....
    ..... 14584, 14588, 14594, 14598
\\_str_convert_lowercase_-
    alphanum:n ... 14691, 14723, 14723
\\_str_convert_lowercase_-
    alphanum_loop:N .....
    ..... 14723, 14725, 14729, 14747
\\_str_convert_pdfname:n .....
    ..... 15745, 15748, 15752, 15779
\\_str_convert_pdfname_bytes:n ..
    ..... 15745, 15755, 15758
\\_str_convert_pdfname_bytes_-
    aux:n ..... 15745, 15760, 15763
\\_str_convert_pdfname_bytes_-
    aux:nnn ..... 15745
\\_str_convert_pdfname_bytes_-
    aux:nnnn ..... 15764, 15765
\\_str_convert_unescape_: .....
    ..... 14773, 14779, 14787, 14788

\\_str_convert_unescape_bytes: ..
    ..... 14773, 14788
\\_str_convert_unescape_hex: ...
    ..... 14998, 14998
\\_str_convert_unescape_name: ...
    ..... 797, 15044
\\_str_convert_unescape_string: .
    ..... 15094, 15099
\\_str_convert_unescape_url: . 15044
\\_str_count:n ..... 772,
    14069, 14129, 14211, 14222, 16715,
    16722, 16733, 16743, 16754, 17006
\\_str_count_aux:n .....
    .. 14211, 14215, 14224, 14229, 14233
\\_str_count_loop:NNNNNNNN 14211,
    14218, 14225, 14230, 14244, 14249
\\_str_count_spaces_loop:w .....
    ..... 14191, 14198, 14204, 14209
\\_str_declare_eight_bit_-
    aux:NNnnn ..... 14848, 14855, 14858
\\_str_declare_eight_bit_-
    encoding:nnnn .....
    ..... 791, 14848, 14848, 15782,
    15789, 15853, 15895, 15952, 16053,
    16140, 16226, 16300, 16313, 16366,
    16464, 16527, 16565, 16580, 41530
\\_str_declare_eight_bit_loop:Nn
    ..... 14848, 14866, 14890, 14894
\\_str_declare_eight_bit_-
    loop:Nnn 14848, 14863, 14884, 14888
\\_str_decode_clist_char:n .....
    ..... 14831, 14837, 14840
\\_str_decode_eight_bit_aux:n ...
    ..... 14896, 14910, 14914
\\_str_decode_eight_bit_aux:Nn ..
    ..... 14896, 14900, 14907
\\_str_decode_native_char:N ....
    ..... 14791, 14792, 14793
\\_str_decode_utf_viii_aux:wNnnwN
    ..... 15338, 15381, 15393
\\_str_decode_utf_viii_continuation:wwN
    ..... 15338, 15366, 15373, 15409
\\_str_decode_utf_viii_end: ....
    ..... 15338, 15348, 15423
\\_str_decode_utf_viii_overflow:w
    ..... 15338, 15407, 15416
\\_str_decode_utf_viii_start:N ..
    ..... 15338, 15347, 15353,
    15371, 15374, 15391, 15394, 15414
\\_str_decode_utf_xvi:Nw .....
    ..... 810, 15519, 15520,
    15522, 15531, 15534, 15535, 15538
\\_str_decode_utf_xvi_bom:NN ...
    ..... 15519, 15525, 15528

```

```

\__str_decode_utf_xvi_error:nnN .
    ..... 15553,
    15571, 15590, 15599, 15604, 15605
\__str_decode_utf_xvi_extra:NNw .
    ..... 15553, 15561, 15603
\__str_decode_utf_xvi_pair:NN ...
    ..... 810, 811, 15547,
    15553, 15553, 15565, 15568, 15592
\__str_decode_utf_xvi_pair_-
    end:Nw .. 15553, 15556, 15572, 15594
\__str_decode_utf_xvi_quad:NNwNN
    ..... 15553, 15560, 15567
\__str_decode_utf_xxxii:Nw .....
    ..... 814, 15677, 15678,
    15680, 15689, 15692, 15693, 15696
\__str_decode_utf_xxxii_bom:NNNN
    ..... 15677, 15683, 15686
\__str_decode_utf_xxxii_end:w ...
    ..... 15677, 15713, 15733
\__str_decode_utf_xxxii_loop:NNNN
    ..... 15677, 15704, 15710, 15731
\__str_encode_clist_char:n .....
    ..... 14842, 14844, 14847
\__str_encode_eight_bit_aux:NNn .
    ..... 14923, 14928, 14936
\__str_encode_eight_bit_aux:nnN .
    ..... 14923, 14938, 14946
\__str_encode_native_char:n ....
    .. 14795, 14802, 14803, 14810, 14814
\__str_encode_utf_vii_loop:wwnnw 803
\__str_encode_utf_viii_char:n ...
    ..... 15262, 15263, 15264
\__str_encode_utf_viii_loop:wwnnw
    ..... 15262, 15266, 15273, 15280
\__str_encode_utf_xvi_aux:N .....
    .. 15434, 15436, 15440, 15442, 15443
\__str_encode_utf_xvi_be:nn ... 808
\__str_encode_utf_xvi_char:n ...
    ..... 15434, 15447, 15450
\__str_encode_utf_xxxii_be:n ...
    ..... 15617, 15619, 15623, 15626
\__str_encode_utf_xxxii_be_-
    aux:nn ..... 15617, 15628, 15631
\__str_encode_utf_xxxii_le:n ...
    ..... 15617, 15625, 15637
\__str_encode_utf_xxxii_le_-
    aux:nn ..... 15617, 15639, 15642
__str_end ..... 15468, 15648
\l__str_end_flag .....
    .... 15470, 15485, 15512, 15543,
    15649, 15656, 15670, 15699, 15737
\g__str_error_bool ..... 14486,
    14623, 14633, 14637, 14642, 14646
\l__str_error_flag .....
    .... 14487, 14809, 14811, 14817,
    14903, 14905, 14917, 14932, 14934,
    14950, 15001, 15012, 15021, 15034,
    15050, 15059, 15072, 15077, 15103,
    15118, 15158, 15340, 15351, 15362,
    15386, 15400, 15420, 15427, 15445,
    15448, 15457, 15540, 15551, 15607,
    15700, 15708, 15718, 15723, 15738
\__str_escape_hex_char:N .....
    ..... 15182, 15183, 15184
\__str_escape_name_char:n .....
    .. 15186, 15189, 15190, 15756, 15779
\c__str_escape_name_not_str ....
    ..... 800, 15186
\c__str_escape_name_str .. 800, 15186
\__str_escape_string_char:N ....
    ..... 15209, 15212, 15213
\c__str_escape_string_str .... 15209
\__str_escape_url_char:n .....
    ..... 15241, 15242, 15243
__str_extra ..... 15285, 15468
\l__str_extra_flag .....
    .... 15286, 15295, 15318, 15342,
    15361, 15469, 15484, 15507, 15542
\__str_filter_bytes:n ... 14749,
    14755, 14772, 14783, 15064, 15126
\__str_filter_bytes_aux:N .....
    ..... 14749, 14757, 14761, 14769
\__str_format_align<:nnnN 835, 16710
\__str_format_align=:nnnN 836, 16749
\__str_format_align>:nnnN 835, 16719
\__str_format_align^:nnnN 836, 16726
\__str_format_fp:nn .....
    ..... 841, 16892, 16893, 16893
\__str_format_fp:NNnnnnN .....
    ..... 841, 16897, 16902, 16902
\__str_format_fp:wnnnNNn .....
    ..... 842, 16928, 16929,
    16930, 16931, 16936, 16940, 16940
\__str_format_fp_e:nn .....
    ..... 16928, 16951, 16951
\__str_format_fp_e_aux:nn .....
    ..... 16951, 16953, 16957
\__str_format_fp_e_aux:wnn .....
    ..... 16951, 16965, 16970
\__str_format_fp_f:nn .....
    ..... 16929, 16985, 16985
\__str_format_fp_f_aux:nn .....
    ..... 16987, 16991
\__str_format_fp_f_aux:wwnn ....
    ..... 16985, 16999, 17003
\__str_format_fp_g:nn .....
    .. 16930, 16931, 16936, 17008, 17008

```

```

__str_format_fp_g_aux:nn .....
    ..... 17012, 17017
__str_format_fp_g_aux:wn .... 17008
__str_format_fp_round:nn .....
    ..... 16949, 16949, 16954, 17013
__str_format_fp_to_scientific:n
    ..... 17008
__str_format_fp_trim:w .....
    ..... 17008, 17029, 17034, 17034
__str_format_fp_trim_dot:w ....
    ..... 17034, 17037, 17040
__str_format_fp_trim_end:w ....
    ..... 17034, 17040, 17041
__str_format_fp_trim_loop:w ...
    ..... 17034, 17036, 17037, 17039
__str_format_if_digit:N ..... 16631
__str_format_if_digit:NTF .....
    ..... 16631, 16682, 16694
__str_format_if_in:nN ..... 16638
__str_format_if_in:nNTF .....
    ..... 16638, 16659, 16668, 16674
__str_format_if_in_aux:NN .....
    ..... 16638, 16640, 16644, 16650
__str_format_int:nn .....
    ..... 839, 16837, 16838, 16838
__str_format_int:NNnnNn .....
    ..... 839, 16842, 16847, 16847
__str_format_int:NwnnNn .....
    ..... 840, 16870, 16871, 16872,
    16873, 16874, 16879, 16883, 16883
__str_format_parse:n ... 16652,
    16652, 16766, 16817, 16843, 16898
__str_format_parse_auxi:NN ....
    ..... 16652, 16654, 16657
__str_format_parse_auxii:nN ...
    ..... 16652, 16662, 16666
__str_format_parse_auxiii:nN ...
    .. 16652, 16660, 16669, 16670, 16672
__str_format_parse_auxiv:nwN ...
    .. 16652, 16677, 16678, 16680, 16683
__str_format_parse_auxv:nN ....
    ..... 16652, 16684, 16686
__str_format_parse_auxvi:nwN ...
    ..... 16652, 16689, 16692, 16695
__str_format_parse_auxvii:nN ...
    ..... 16652, 16690, 16696, 16698
__str_format_parse_end:nwn ....
    ..... 16652, 16701, 16702, 16704
__str_format_put:nw .... 16636,
    16636, 16637, 16776, 16780, 16781,
    16788, 16790, 16791, 16850, 16851,
    16853, 16857, 16858, 16860, 16862,
    16906, 16907, 16909, 16913, 16914,
    16916, 16918, 16922, 16923, 16925,
    16972, 16975, 16976, 16979, 16981
__str_format_seq:nn .....
    ..... 16815, 16821, 16821, 16827
__str_format_seq_end:w .....
    ..... 16825, 16835, 16835
__str_format_seq_loop:nnNn ....
    .... 839, 16823, 16828, 16828, 16831
__str_format_tl:NNnnNn .....
    .... 837, 16765, 16770, 16770, 16832
__str_format_tl_s:NNnnNn .....
    ..... 838, 16800, 16803, 16803
__str_head:w 773, 14251, 14255, 14259
__str_hexadecimal_use:N ..... 14521
__str_hexadecimal_use:NTF .....
    797, 14521, 15018, 15028, 15067, 15069
__str_if_contains_char:Nn ... 14489
__str_if_contains_char:nn ... 14498
__str_if_contains_char:NnTF ...
    ..... 14489, 15198, 15204, 15217
__str_if_contains_char:nNTF .. 833
__str_if_contains_char:nnTF ...
    ..... 781, 14489, 15251, 15257
__str_if_contains_char_aux:nn ..
    ..... 14489, 14491, 14496
__str_if_contains_char_auxi:nN .
    .. 14489, 14497, 14500, 14504, 14509
__str_if_contains_char_true: ...
    ..... 14489, 14507, 14511
__str_if_eq:nn 762, 13816, 13816,
    13820, 13826, 13831, 13838, 13843
__str_if_escape_name:n ..... 15195
__str_if_escape_name:nTF .....
    ..... 15186, 15192
__str_if_escape_string:N .... 15229
__str_if_escape_string:NTF .....
    ..... 15209, 15215
__str_if_escape_url:n ..... 15248
__str_if_escape_url:nTF 15241, 15245
__str_if_flag_error:Nnn .....
    ..... 784, 785, 14616, 14616,
    14635, 14644, 14784, 14811, 14905,
    14934, 15012, 15058, 15059, 15117,
    15118, 15351, 15448, 15551, 15708
__str_if_flag_no_error:Nnn ....
    .... 784, 14616, 14622, 14635, 14644
__str_if_flag_times:NTF . 14624,
    14624, 15294, 15295, 15296, 15297,
    15483, 15484, 15485, 15655, 15656
__str_if_recursion_tail_-
    break:NN 13689, 13689, 13969, 13987
__str_if_recursion_tail_stop_-
    do:Nn ..... 13689, 13690, 14317

```



```

\__str_item:nn .....
.... 768, 14053, 14059, 14064, 14065
\__str_item:w 768, 14053, 14067, 14072
\__str_map_function:nn .....
765, 13929, 13932, 13941, 13946, 14000
\__str_map_function:w .....
765, 13929, 13931, 13939, 13940, 14000
\__str_map_inline:NN .....
..... 13929, 13956, 13967, 13971
\__str_map_variable:NnN .....
..... 13929, 13977, 13985, 13990
\c__str_max_byte_int .. 14456, 14816
__str_missing ..... 15285, 15468
\l__str_missing_flag .....
15285, 15294, 15312, 15341, 15385,
15426, 15468, 15483, 15502, 15541
\l__str_modulo_int ..... 14923
\__str_octal_use:N ..... 14513
\__str_octal_use:NTF .....
781, 782, 14513, 15129, 15131, 15133
\__str_output_byte:n ..... 812,
14552, 14552, 14581, 14582, 14744,
14949, 15277, 15283, 15635, 15644
\__str_output_byte:w .....
..... 797, 14552, 14553,
14554, 15005, 15031, 15066, 15128
\__str_output_byte_pair:nnN ....
..... 14568, 14570, 14575, 14578
\__str_output_byte_pair_be:n ...
.. 14568, 14568, 15436, 15440, 15634
\__str_output_byte_pair_le:n ...
..... 14568, 14573, 15442, 15645
\__str_output_end: .....
..... 797, 14552, 14553, 14566,
15010, 15030, 15080, 15162, 15166
\__str_output_hexadecimal:n ....
.... 14552, 14560, 15185, 15193,
15246, 15767, 15768, 15771, 15774
__str_overflow ..... 15285, 15648
\l__str_overflow_flag ... 15288,
15297, 15330, 15344, 15419, 15648,
15655, 15663, 15698, 15717, 15722
__str_overlong ..... 15285
\l__str_overlong_flag .....
.. 15287, 15296, 15323, 15343, 15399
\__str_range:nnn ..... 837, 14114,
14120, 14125, 14126, 16790, 16791
\__str_range:nnw . 14114, 14136, 14140
\__str_range:w .. 14114, 14128, 14134
\__str_range_normalize:nn .....
..... 14137, 14138, 14146, 14146
\__str_replace:NNNnn .... 13754,
13755, 13757, 13759, 13761, 13766
\__str_replace_aux:NNNnnn .....
..... 13754, 13775, 13781
\__str_replace_next:w ... 13754,
13759, 13761, 13783, 13786, 13793
\c__str_replacement_char_int ...
..... 14455, 14918,
15363, 15387, 15401, 15421, 15428,
15458, 15610, 15719, 15724, 15740
\g__str_result_tl ..... 779, 783-
785, 789, 791, 797, 810, 812, 814,
14454, 14586, 14590, 14602, 14606,
14652, 14663, 14664, 14666, 14782,
14783, 14833, 14834, 14837, 14845,
15003, 15007, 15052, 15054, 15105,
15108, 15111, 15114, 15345, 15347,
15437, 15520, 15522, 15526, 15545,
15620, 15678, 15680, 15683, 15702
\__str_sep: .. 14052, 14052, 14068,
14069, 14072, 14081, 14089, 14093,
14101, 14103, 14106, 14108, 14129,
14130, 14131, 14134, 14143, 14144,
14165, 14166, 14167, 14174, 14176,
14179, 14182, 15266, 15274, 15281
\__str_skip_end:NNNNNNNN .....
..... 769, 14093, 14110, 14113
\__str_skip_end:w 14093, 14098, 14108
\__str_skip_exp_end:w .....
..... 769, 771, 14080,
14089, 14093, 14093, 14105, 14144
\__str_skip_loop:wNNNNNNNN .....
..... 14093, 14096, 14103
\__str_tail_auxi:w 14266, 14270, 14274
\__str_tail_auxii:w .....
..... 774, 14266, 14277, 14280
\__str_tmp:n .... 13692, 13698, 13701
\__str_tmp:w ..... 793, 797,
808, 810, 814, 14452, 14452, 14898,
14904, 14926, 14933, 15044, 15090,
15092, 15097, 15122, 15446, 15453,
15458, 15460, 15463, 15464, 15544,
15559, 15564, 15575, 15578, 15584,
15585, 15701, 15716, 15721, 15727
\l__str_tmp_tl . 787, 14452, 14541,
14542, 14544, 14699, 14700, 14701,
14703, 14707, 14711, 14718, 14850
\__str_to_other_end:w .....
..... 767, 14007, 14022, 14027
\__str_to_other_fast_end:w .....
..... 14030, 14045, 14050
\__str_to_other_fast_loop:w ....
..... 14032, 14041, 14048
\__str_to_other_loop:w .....
.... 767, 14007, 14009, 14018, 14024

```


- _str_unescape_hex_auxi:N
- .. [14998](#), [15006](#), [15015](#), [15022](#), [15031](#)
- _str_unescape_hex_auxii:N
- [14998](#), [15019](#), [15025](#), [15035](#)
- _str_unescape_name_loop:wNN
- [15044](#), [15091](#)
- _str_unescape_string_loop:wNNN
- .. [15094](#), [15113](#), [15124](#), [15163](#), [15166](#)
- _str_unescape_string_newlines:wN
- [15094](#), [15107](#), [15167](#), [15171](#)
- _str_unescape_string_repeat:NNNNN
- .. [15094](#), [15138](#), [15140](#), [15142](#), [15165](#)
- _str_unescape_url_loop:wNN
- [15044](#), [15093](#)
- _str_use_i_delimit_by_s_-
- stop:nw [773](#), [13685](#), [13686](#),
- [14079](#), [14088](#), [14207](#), [14260](#), [14263](#)
- _str_use_none_delimit_by_s_-
- stop:w [13685](#),
- [13685](#), [13788](#), [14077](#), [14086](#), [14247](#),
- [14612](#), [14886](#), [14892](#), [15278](#), [15370](#)
- \strcmp [5](#)
- \string [416](#)
- \sum [1499](#)
- \suppressfontnotfounderror [703](#)
- \suppressifcsnameerror [916](#)
- \suppresslongerror [917](#)
- \suppressmathparerror [918](#)
- \suppressoutererror [919](#)
- \suppressprimitiveerror [920](#)
- \synctex [683](#)
- sys commands:
- \c_sys_backend_str [82](#), [8863](#), [8935](#)
- \c_sys_day_int [76](#), [9071](#)
- \c_sys_engine_exec_str
- [77](#), [618](#), [8754](#), [11721](#)
- \c_sys_engine_format_str
- [77](#), [618](#), [8754](#), [11722](#)
- \c_sys_engine_str
- [77](#), [618](#), [699](#), [8734](#), [8808](#)
- \c_sys_engine_version_str [78](#), [8806](#)
- \sys_ensure_backend: [78](#), [81](#), [8933](#), [8933](#)
- \sys_finalise: [41543](#), [41544](#)
- \sys_finalize:
- .. [82](#), [8865](#), [9236](#), [9236](#), [41543](#), [41544](#)
- \sys_get_query:nN [81](#), [9159](#), [9159](#)
- \sys_get_query:nnN [81](#), [9159](#), [9161](#)
- \sys_get_query:nnnN
- .. [81](#), [9159](#), [9160](#), [9162](#), [9163](#), [9229](#)
- \sys_get_shell:nnN
- [79](#), [8957](#), [8957](#), [8962](#), [9188](#)
- \sys_get_shell:nnNTF [79](#), [94](#), [8957](#), [8959](#)
- \sys_gset_rand_seed:n
- [79](#), [286](#), [9116](#), [9118](#)
- \c_sys_hour_int [76](#), [9071](#)
- \sys_if_engine luatex:TF
- .. [77](#), [108](#), [8734](#), [8762](#), [8787](#), [8901](#),
- [8911](#), [8912](#), [8997](#), [9019](#), [9051](#), [9134](#),
- [9142](#), [10327](#), [11160](#), [11373](#), [11525](#),
- [11806](#), [11853](#), [20365](#), [31517](#), [31559](#),
- [31604](#), [31617](#), [31643](#), [31712](#), [31736](#),
- [31773](#), [32080](#), [32129](#), [32508](#), [32588](#),
- [32596](#), [32613](#), [32644](#), [40949](#), [41014](#)
- \sys_if_engine luatex_p:
- [77](#), [8734](#), [10513](#),
- [14477](#), [14751](#), [14775](#), [14797](#), [14967](#)
- \sys_if_engine_opentype:TF [77](#),
- [618](#), [15750](#), [31827](#), [31852](#), [31915](#),
- [32102](#), [32823](#), [32861](#), [33868](#), [34930](#)
- \sys_if_engine_opentype_p: [77](#), [618](#)
- \sys_if_engine_pdftex:TF [77](#),
- [8734](#), [8758](#), [8782](#), [9269](#), [32832](#), [32882](#)
- \sys_if_engine_pdftex_p:
- [77](#), [8734](#), [8796](#)
- \sys_if_engine_ptex:TF
- [77](#), [8734](#), [8760](#), [8785](#), [33834](#)
- \sys_if_engine_ptex_p:
- [77](#), [3676](#), [3698](#), [8734](#)
- \sys_if_engine_uptex:TF
- [77](#), [8734](#), [8761](#), [8786](#)
- \sys_if_engine_uptex_p:
- [77](#), [3677](#), [3699](#), [8734](#)
- \sys_if_engine_xetex:TF
- .. [7](#), [77](#), [8734](#), [8759](#), [8784](#), [8882](#), [9264](#)
- \sys_if_engine_xetex_p:
- [77](#), [8734](#), [9082](#),
- [14478](#), [14752](#), [14776](#), [14798](#), [14968](#)
- \sys_if_output_dvi:TF [78](#), [9245](#)
- \sys_if_output_dvi_p: [78](#), [9245](#)
- \sys_if_output_pdf:TF
- [78](#), [8897](#), [9245](#), [9267](#)
- \sys_if_output_pdf_p: [78](#), [9245](#)
- \sys_if_platform_unix:TF
- [79](#), [8863](#), [11824](#)
- \sys_if_platform_unix_p:
- [79](#), [8863](#), [11824](#)
- \sys_if_platform_windows:TF
- [79](#), [8863](#), [11824](#)
- \sys_if_platform_windows_p:
- [79](#), [8863](#), [11824](#)
- \sys_if_shell:TF
- .. [80](#), [8964](#), [9150](#), [9186](#), [10333](#), [10583](#)
- \sys_if_shell_p: [80](#), [9150](#)
- \sys_if_shell_restricted:TF
- [80](#), [9150](#), [9176](#), [9178](#)
- \sys_if_shell_restricted_p: [80](#), [9150](#)
- \sys_if_shell_unrestricted:TF
- [80](#), [9150](#)

- \sys_if_shell_unrestricted_p: ... [80](#), [9150](#)
- \sys_if_timer_exist:TF ... [9121](#), [41547](#), [41548](#), [41550](#), [41552](#)
- \sys_if_timer_exist_p: ... [9121](#), [41547](#)
- \c_sys_jobname_str . [76](#), [102](#), [630](#), [9069](#)
- \sys_load_backend:n ... [78](#), [81](#), [82](#), [8863](#), [8863](#), [8936](#)
- \sys_load_debug: ... [82](#), [1565](#), [1571](#), [8939](#), [8939](#), [8951](#), [10200](#)
- \sys_load_deprecation: . [41545](#), [41546](#)
- \c_sys_minute_int ... [76](#), [9071](#)
- \c_sys_month_int ... [76](#), [9071](#)
- \c_sys_output_str ... [78](#), [9245](#)
- \c_sys_platform_str ... [79](#), [8863](#), [11806](#), [11827](#)
- \sys_rand_seed: [79](#), [163](#), [286](#), [9112](#), [9114](#)
- \c_sys_shell_escape_int ... [80](#), [9138](#), [9153](#), [9155](#), [9157](#)
- \sys_shell_now:n ... [80](#), [9000](#), [9021](#), [9025](#), [9028](#)
- \sys_shell_shipout:n ... [80](#), [9030](#), [9053](#), [9057](#), [9060](#)
- \sys_split_query:nN . [81](#), [9217](#), [9217](#)
- \sys_split_query:nnN . [81](#), [9217](#), [9219](#)
- \sys_split_query:nnnN ... [81](#), [9217](#), [9218](#), [9220](#), [9226](#)
- \sys_timer: ... [78](#), [1545](#), [9121](#), [9132](#), [41188](#), [41194](#), [41344](#), [41357](#)
- \c_sys_timestamp_str ... [76](#), [9103](#)
- \c_sys_year_int ... [76](#), [9071](#)
- sys internal commands:
 - \g__sys_backend_tl ... [8873](#), [8874](#), [8875](#), [9259](#)
 - __sys_const:nn . [8718](#), [8718](#), [8748](#), [8751](#), [9152](#), [9154](#), [9156](#), [9254](#), [9256](#)
 - \g__sys_debug_bool . [8938](#), [8941](#), [8943](#)
 - __sys_elapsedtime: ... [9121](#), [9135](#)
 - __sys_everyjob:n ... [9061](#), [9066](#), [9069](#), [9071](#), [9103](#), [9112](#), [9116](#), [9138](#), [9150](#), [9234](#)
 - \g__sys_everyjob_tl ... [9061](#)
 - __sys_finalize:n ... [9236](#), [9242](#), [9245](#), [9260](#), [9277](#)
 - \g__sys_finalize_tl ... [9236](#)
 - __sys_get:nnN ... [8957](#), [8965](#), [8968](#)
 - __sys_get_do:Nw ... [8957](#), [8982](#), [8991](#)
 - __sys_get_query:Nw ... [9203](#), [9214](#)
 - __sys_get_query_auxi:nnnN ... [9159](#), [9166](#), [9168](#), [9183](#)
 - __sys_get_query_auxii:nnnN ... [9159](#), [9170](#), [9184](#), [9208](#)
 - __sys_load_backend_check:N ... [8863](#), [8874](#), [8880](#)
- \c__sys_marker_tl ... [8956](#), [8980](#), [8992](#)
- __sys_shell_now:n ... [9000](#), [9022](#)
- __sys_shell_shipout:n ... [9030](#), [9054](#)
- \c__sys_shell_stream_int ... [8997](#), [9026](#), [9058](#)
- __sys_tmp:w ... [9074](#), [9095](#), [9097](#), [9098](#), [9099](#), [9100](#)
- \l__sys_tmp_tl ... [8717](#), [9200](#), [9201](#), [9204](#), [9229](#), [9230](#), [9231](#)
- syst commands:
 - \c_syst_catcodes_n ... [31779](#), [31783](#)
 - \c_syst_last_allocated_toks . [3255](#)
- T**
- \T ... [65](#)
- \t ... [33333](#), [35799](#), [35825](#)
- \tabskip ... [417](#)
- \tagcode ... [684](#)
- tan ... [283](#)
- tand ... [284](#)
- \tate ... [1187](#)
- \tbaselineshift ... [1188](#)
- \TeX ... [40525](#)
- TeX and L^AT_EX 2_ε commands:
 - \@ ... [14418](#)
 - \@@@hyph ... [376](#)
 - \@@end ... [1222](#), [1223](#)
 - \@@hyph ... [1226](#), [1229](#)
 - \@@input ... [1224](#)
 - \@@italiccorr ... [1230](#)
 - \@@shipout ... [1232](#), [1233](#)
 - \@@tracingfonts ... [377](#), [1268](#)
 - \@@underline ... [1231](#)
 - \@addtofilelist ... [11521](#), [40644](#)
 - \@changed@cmd ... [33265](#), [35699](#)
 - \@classoptionslist . [9279](#), [9281](#), [9283](#)
 - \@current@cmd ... [33264](#), [35698](#)
 - \@currnamestack ... [680](#), [11026](#), [11028](#), [11029](#)
 - \@expl@finalise@setup@@ [8945](#), [8947](#), [31819](#), [31820](#), [32949](#), [32951](#), [34894](#), [34896](#), [40816](#), [40818](#), [40876](#), [40878](#)
 - \@expl@luadata@bytecode ... [19](#)
 - \@filelist ... [107](#), [681](#), [693](#), [696](#), [697](#), [11520](#), [11636](#), [11639](#), [11648](#), [11653](#), [40643](#)
 - \@firstofone ... [25](#)
 - \@firstoftwo ... [395](#)
 - \@gobble ... [27](#)
 - \@gobbletwo ... [27](#)
 - \@kernel@after@begindocument . [8949](#), [32953](#), [34898](#), [40880](#)
 - \@kernel@before@begindocument . [41133](#), [41135](#)

- \@protected@testopt 1350, 33252
- \@secondoftwo 395
- \@sptoken 206
- \@tempa 1240, 1254, 1257
- \@tfor 377, 1240
- \@uclclist 1386, 34923
- \@unexpandable@protect 1107
- \@unusedoptionlist 9298
- \active@prefix 33111
- \afterassignment 584, 1429
- \aftergroup 15
- \AtBeginDocument 376
- \begingroup 14
- \bgroup 206
- \botmark 945
- \box 319
- \catcodetable 1307, 1311, 1313
- \char 218
- \chardef
 - . . . 209, 210, 604, 607, 884, 1339, 1348
- \check@mathfonts 317
- \conditionally@traceoff
 - 673, 9701, 10746
- \conditionally@traceon 9719
- \copy 312
- \count 217, 449
- \cr 615
- \CROP@shipout 1241
- \csname 23, 383, 682, 683
- \csstring 405
- \currentgrouplevel . . . 418, 646, 1310
- \currentgrouptype 418, 646
- \day 76
- \declare@file@substitution . . . 40819
- \def 217
- \detokenize 118
- \development@branch@name 11724, 11725
- \dimen 943
- \dimendef 943
- \dimexpr 1422
- \directlua 108
- \dp 312, 1108, 1109
- \dup@shipout 1242
- \e@alloc@ccodetable@count 31777
- \e@alloc@top 449, 3241
- \edef 3, 6, 714
- \egroup 206
- \else 29
- \end 376, 639
- \endcsname 23
- \endgroup 14
- \endinput 87
- \endlinechar 96, 129, 130, 296–
 - 298, 720–723, 945, 1307, 1308, 1310
- \endtemplate 75, 615
- \errhelp 635
- \errmessage 635, 636
- \errorcontextlines 385, 636, 751, 1425
- \escapechar 118, 405, 419, 491, 671
- \everyeof 722
- \everyjob 625
- \everyoef 723
- \everypar 31, 213, 423
- \expandafter 42, 43
- \expanded
 - . . . 3, 6, 27, 36, 352, 425, 428, 441, 721
- \fi 29, 216
- \firstmark 435, 945
- \fmtname 77
- \font 216, 942
- \fontdimen 63, 264, 1056–1059
- \frozen@everydisplay 1227
- \frozen@everymath 1228
- \futurelet
 - 464, 467, 469, 480, 615, 950, 953
- \global 355
- \GPTorg@shipout 1243
- \halign 75, 106, 342, 423, 615, 935
- \hskip 241
- \ht 313, 1108, 1109
- \hyphen 945
- \hyphenchar 1056
- \if 30
- \ifcase 185
- \ifcat 30
- \ifcsname 30
- \ifdefined 30
- \ifdim 244
- \ifeof 102
- \iffalse 29, 68, 712
- \ifhbox 323
- \ifhmode 30
- \ifincsname 352
- \ifinner 30
- \ifmmode 30, 712
- \ifnum 185
- \ifodd 186, 954
- \iftrue 29, 68, 712
- \ifvbox 323
- \ifvmode 30
- \ifvoid 323
- \ifx 30
- \indent 423
- \infty 279
- \input 106, 376
- \input@path 103, 685, 11207, 11209
- \italiccorr 945
- \jobname 76, 625

- \kcatcode 301
- \kern 238
- \lastnamedcs 408
- \lccode 467, 472, 904
- \leavevmode 31
- \let 355, 482
- \latcharcode 933
- \LL@shipout 1244
- \loctoks 449
- \long 5, 218, 752
- \lower 1444
- \lowercase 483, 572, 573
- \luaescapestring 109
- \makeatletter 10
- \mathchar 218
- \mathchardef 210, 884, 1339
- \mathop 1497
- \mathord 335
- \maxdimen 236
- \meaning 22, 206, 217,
218, 420, 465, 467, 942, 943, 953, 954
- \mem@oldshipout 1245
- \message 36
- \month 76
- \newcatcodetable 1307
- \newif 68, 111
- \newlinechar 129,
130, 385, 409, 636, 669, 720–722, 751
- \newread 659
- \newtoks 46, 460, 490
- \newwrite 666
- \noexpand 42, 216
- \nullfont 945
- \number 185, 881, 1165
- \numexpr 384
- \opem@shipout 1246
- \or 185
- \outer 8, 218, 464, 481,
482, 659, 666, 935, 954, 1552, 1553
- \parindent 31
- \pdfescapehex 796
- \pdfescapename 147, 796
- \pdfescapestring 147, 796
- \pdffeedback 626
- \pdffilesize 685
- \pdfmapfile 377
- \pdfmapline 377
- \pdfstrcmp 351, 366, 762
- \pdfuniformdeviate 286
- \pgfpages@originalshipout 1247
- \pi 279
- \pr@shipout 1248
- \primitive 377, 626
- \protect 673, 1106, 1107, 1404
- \protected 218, 752
- \protected@edef 1344
- \ProvidesClass 10
- \ProvidesFile 10
- \ProvidesPackage 10
- \quitvmode 423
- \read 96, 664
- \readline 96, 664
- \relax 29, 30, 216, 401,
406, 419, 482, 608, 682, 905, 907,
960, 967, 969, 1063, 1065, 1090, 1122
- \RequirePackage 11, 680
- \romannumeral ... 44, 1063, 1338, 1345
- \savecatcodetable 1309
- \scantextokens 723
- \scantokens 130, 149, 684, 720, 723, 725
- \shipout 376
- \show 22, 99, 121, 419
- \showbox 1425
- \showgroups 15, 419
- \showstream 99, 669
- \showthe ... 418, 903, 998, 1003, 1005
- \showtokens 99, 121, 641, 751
- \sin 279
- \skip 473, 474
- \space 945
- \splitbotmark 945
- \splitfirstmark 945
- \SS 1406
- \strcmp 351, 366
- \string 206, 467, 469, 470
- \tenrm 216
- \tex_show:D 670
- \tex_showstream:D 670
- \tex_showtokens:D 670
- \tex_write:D 670
- \the
175, 216, 235, 240, 243, 427, 881, 1422
- \time 76
- \toks 46, 163, 185,
447–455, 460, 466, 469, 470, 472,
474, 476, 490, 491, 541, 548, 549,
553, 554, 564, 572, 580, 593, 602, 866
- \toksdef 460
- \topmark 217, 945
- \tracingfonts 377
- \tracingnesting 684, 720
- \tracingonline 1425
- \typeout 673
- \Ucharcat 935
- \Umathcode 77
- \undefined 969

- `\unexpanded` 42, 119,
120, 125, 126, 160, 161, 166, 167,
193, 196, 197, 199, 224, 714, 741, 888
- `\unhbox` 319
- `\unhcopy` 316
- `\uniformdeviate` 286
- `\unless` 29
- `\unvbox` 319
- `\unvcopy` 318
- `\uppercase` 572
- `\usepackage` 680
- `\UTFviii@four@octets` 33110
- `\UTFviii@three@octets` 33109
- `\UTFviii@two@octets` 33108
- `\valign` 615
- `\vcenter` 317, 1429
- `\verb` 130
- `\verso@orig@shipout` 1250
- `\vskip` 241
- `\vtop` 1450
- `\wd` 313, 1108, 1109
- `\write` 100, 669
- `\year` 76
- tex commands:
 - `\tex_above:D` 150
 - `\tex_abovedisplayshortskip:D` .. 151
 - `\tex_abovedisplayskip:D` 152
 - `\tex_abovewithdelims:D` 153
 - `\tex_accent:D` 154
 - `\tex_adjdemerits:D` 155
 - `\tex_adjustinterwordglue:D` 627
 - `\tex_adjustspacing:D` 628, 932
 - `\tex_advance:D` .. 156, 3348, 3355,
3358, 3809, 3811, 3844, 3846, 5356,
18361, 18363, 18365, 18367, 18373,
18375, 18377, 18379, 21626, 21629,
21635, 21638, 22094, 22096, 22100,
22102, 22188, 22190, 22194, 22196
 - `\tex_afterassignment:D` . 157, 3750,
3793, 7613, 7677, 7821, 20553, 31364
 - `\tex_aftergroup:D` ... 1313, 158, 1421
 - `\tex_alignmark:D` 779
 - `\tex_aligntab:D` 780
 - `\tex_appendkern:D` 629
 - `\tex_atop:D` 159
 - `\tex_atopwithdelims:D` 160
 - `\tex_attribute:D` 781
 - `\tex_attributedef:D` 782
 - `\tex_automaticdiscretionary:D` .. 784
 - `\tex_automatichyphenmode:D` 785
 - `\tex_automatichyphenpenalty:D` .. 787
 - `\tex_autospacing:D` 1134
 - `\tex_autoxspacing:D` 1135
 - `\tex_badness:D` 161
 - `\tex_baselineskip:D` 162
 - `\tex_batchmode:D` 163, 9565
 - `\tex_begincsname:D` 788
 - `\tex_beginingroup:D` 164, 1235, 1279, 1416
 - `\tex_beginL:D` 472
 - `\tex_beginR:D` 473
 - `\tex_belowdisplayshortskip:D` .. 165
 - `\tex_belowdisplayskip:D` 166
 - `\tex_binoppenalty:D` 167
 - `\tex_bodydir:D` 789
 - `\tex_bodydirection:D` 790
 - `\tex_botmark:D` 168
 - `\tex_botmarks:D` 474
 - `\tex_boundary:D` 791
 - `\tex_box:D` ... 169, 36417, 36419, 36462
 - `\tex_boxdir:D` 792
 - `\tex_boxdirection:D` 793
 - `\tex_boxmaxdepth:D` 170
 - `\tex_breakafterdirmode:D` 794
 - `\tex_brokenpenalty:D` 171
 - `\tex_catcode:D` .. 172, 2743, 7081,
8706, 8709, 12299, 12639, 19869, 19871
 - `\tex_catcodetable:D` 795, 31621, 31628
 - `\tex_char:D` 173
 - `\tex_chardef:D` 393,
174, 1409, 1438, 1440, 1796, 1797,
8373, 8395, 8400, 10324, 10575,
18336, 20434, 32981, 32983, 32985
 - `\tex_cleaders:D` 175
 - `\tex_clearmarks:D` 796
 - `\tex_closein:D` 176, 10357
 - `\tex_closeout:D` 177, 10601
 - `\tex_clubpenalties:D` 475
 - `\tex_clubpenalty:D` 178
 - `\tex_compoundhyphenmode:D` 798
 - `\tex_copy:D` 179, 36411,
36413, 36437, 36446, 36455, 36463
 - `\tex_copyfont:D` 630, 933
 - `\tex_count:D` 180, 3224, 3240,
3248, 3249, 10272, 10274, 10525, 10527
 - `\tex_countdef:D` 181
 - `\tex_cr:D` 182
 - `\tex_crampeddisplaystyle:D` 799
 - `\tex_crampedscriptscriptstyle:D` 801
 - `\tex_crampedscriptstyle:D` 802
 - `\tex_crampedtextstyle:D` 803
 - `\tex_crcr:D` 183
 - `\tex_creationdate:D` .. 631, 768, 9109
 - `\tex_csname:D` 184, 1403
 - `\tex_csstring:D` 804
 - `\tex_currentcjktoken:D` 1136
 - `\tex_currentgrouplevel:D`
..... 1312, 476, 20928, 20946,
31610, 31690, 31700, 31707, 39654

- `\tex_currentgrouptype:D` 477
- `\tex_currentifbranch:D` 478
- `\tex_currentiflevel:D` 479
- `\tex_currentifttype:D` 480
- `\tex_currentspacingmode:D` 1137
- `\tex_currentxspacingmode:D` ... 1138
- `\tex_day:D` 185, 1286, 1290
- `\tex_deadcycles:D` 186
- `\tex_def:D`
 - ... 187, 687, 688, 689, 1466, 1470,
 - 1475, 1480, 42029, 42053, 42087, 42502
- `\tex_defaultthyphenchar:D` 188
- `\tex_defaultskewchar:D` 189
- `\tex_deferred:D` 805
- `\tex_delcode:D` 190
- `\tex_delimiter:D` 191
- `\tex_delimiterfactor:D` 192
- `\tex_delimitershortfall:D` 193
- `\tex_detokenize:D` 481, 1412, 1414
- `\tex_dimen:D` 194
- `\tex_dimendef:D` 195
- `\tex_dimexpr:D` 482, 21581, 22054, 36389
- `\tex_directlua:D` . 808, 1266, 1267,
 - 8755, 9107, 9108, 9144, 11809, 11833
- `\tex_disablecjktoken:D` 1200
- `\tex_discretionary:D` 196
- `\tex_discretionaryligaturemode:D` 807
- `\tex_disinhibitglue:D` 1139
- `\tex_displayindent:D` 197
- `\tex_displaylimits:D` 198, 39301
- `\tex_displaystyle:D` 199
- `\tex_displaywidowpenalties:D` .. 483
- `\tex_displaywidowpenalty:D` 200
- `\tex_displaywidth:D` 201
- `\tex_divide:D` 202, 3218, 5357
- `\tex_doublehyphendemerits:D` ... 203
- `\tex_dp:D` 204, 36427, 36670
- `\tex_draftmode:D` 632, 934
- `\tex_dtou:D` 1140
- `\tex_dump:D` 205
- `\tex_dviextension:D` 809
- `\tex_dvifedback:D` 810
- `\tex_dvivariable:D` 811
- `\tex_eachlinedepth:D` 633
- `\tex_eachlineheight:D` 634
- `\tex_edef:D` 206, 1236,
 - 1237, 1253, 1280, 1281, 1286, 1287,
 - 1292, 1293, 1298, 1299, 1467, 1468,
 - 1472, 1477, 1482, 10839, 10897, 41415
- `\tex_efcode:D` 670
- `\tex_elapsedtime:D` ... 635, 769, 9136
- `\tex_else:D` . 207, 1239, 1265, 1283,
 - 1289, 1295, 1301, 1389, 1441, 1444
- `\tex_emergencystretch:D` 208
- `\tex_enablecjktoken:D` ... 1201, 8740
- `\tex_end:D` 209, 1223, 1312, 1965
- `\tex_endcsname:D` 210, 1404
- `\tex_endgroup:D`
 - 211, 1221, 1261, 1304, 1417
- `\tex_endinput:D` 212, 9574, 11505, 11734
- `\tex_endL:D` 484
- `\tex_endlinechar:D` 120, 121,
 - 134, 213, 9198, 10429, 10431, 10432,
 - 12487, 12488, 12489, 12523, 12614,
 - 12618, 12644, 31566, 31570, 31583,
 - 31623, 31624, 31639, 31810, 31825,
 - 31859, 42012, 42074, 42083, 42499
- `\tex_endlocalcontrol:D` 814
- `\tex_endR:D` 485
- `\tex_epTeXinputencoding:D` 1141
- `\tex_epTeXversion:D` . 1142, 8828, 8853
- `\tex_eqno:D` 214
- `\tex_errhelp:D` 215, 9442
- `\tex_errmessage:D` 216, 1957, 9462
- `\tex_errorcontextlines:D`
 - 217, 2319, 2326,
 - 2327, 9457, 9476, 9663, 13540, 36526
- `\tex_errorstopmode:D` 218
- `\tex_escapechar:D` . 682, 219, 2304,
 - 3715, 3777, 3778, 4130, 4228, 4229,
 - 4232, 4260, 4360, 10451, 10700,
 - 10747, 10753, 15002, 15051, 15104
- `\tex_escapehex:D` 636
- `\tex_escapename:D` 637
- `\tex_escapestring:D` 638
- `\tex_eTeXglueshrinkorder:D` 812
- `\tex_eTeXgluestretchorder:D` ... 813
- `\tex_eTeXrevision:D` 486
- `\tex_eTeXversion:D` 487
- `\tex_etoksapp:D` 815, 4382, 4383
- `\tex_etokspre:D` 816, 4376, 4377
- `\tex_euc:D` 1143
- `\tex_everycr:D` 220
- `\tex_everydisplay:D` 221, 1227
- `\tex_everyeof:D`
 - 488, 8980, 11153, 12496, 12548
- `\tex_everyhbox:D` 222
- `\tex_everyjob:D` 223, 1306, 1313
- `\tex_everymath:D` 224, 1228
- `\tex_everypar:D` 225
- `\tex_everyvbox:D` 226
- `\tex_exceptionpenalty:D` 817
- `\tex_exhyphenchar:D` 818
- `\tex_exhyphenpenalty:D` 227
- `\tex_expandafter:D`
 - 228, 692, 1240, 1254, 1256, 1257, 1405
- `\tex_expanded:D` 441, 820,
 - 1322, 1493, 2458, 2521, 2550, 2608,

- 2647, 2664, 2729, 2953, 4380, 4386,
 12338, 12369, 12410, 12438, 12624,
 13150, 21460, 22230, 22570, 31983
 \tex_explicitdiscretionary:D .. 821
 \tex_explicithyphenpenalty:D .. 819
 \tex_fam:D 229
 \tex_fi:D 230,
 693, 1225, 1234, 1258, 1260, 1269,
 1270, 1271, 1273, 1274, 1278, 1285,
 1291, 1297, 1303, 1307, 1310, 1323,
 1329, 1334, 1390, 1446, 1447, 36666
 \tex_filedump:D
 . 702, 770, 1377, 11365, 11378, 11385
 \tex_filemoddate:D
 701, 771, 1373, 11484
 \tex_filesize:D
 ... 688, 699, 772, 1360, 11172, 11385
 \tex_finalhyphendemerits:D 231
 \tex_firstlineheight:D 639
 \tex_firstmark:D 232
 \tex_firstmarks:D 489
 \tex_firstvalidlanguage:D 822
 \tex_fixupboxesmode:D 824
 \tex_floatingpenalty:D 233
 \tex_font:D 234, 24015
 \tex_fontchardp:D 490
 \tex_fontcharht:D 491
 \tex_fontcharic:D 492
 \tex_fontcharwd:D 493
 \tex_fontdimen:D 235, 24007
 \tex_fontexpand:D 640, 935
 \tex_fontid:D 825
 \tex_fontname:D 236
 \tex_fontsize:D 641
 \tex_forcecjktoken:D 1202
 \tex_formatname:D 826
 \tex_futurelet:D . 237, 3723, 3781,
 4234, 4247, 4308, 4331, 20548, 20550
 \tex_gdef:D 238, 1422, 1424,
 1426, 1427, 1448, 1451, 1452, 1453,
 1456, 1457, 1458, 1461, 1462, 1463
 \tex_gleaders:D 832
 \tex_glet:D 833
 \tex_global:D 986, 140, 144, 239, 694,
 1256, 1284, 1290, 1296, 1302, 1386,
 1387, 1388, 1389, 1390, 1391, 1392,
 1393, 1394, 1395, 1396, 1397, 1398,
 1399, 1400, 1401, 1402, 1403, 1404,
 1405, 1406, 1407, 1408, 1409, 1410,
 1411, 1412, 1413, 1414, 1415, 1416,
 1417, 1419, 1420, 1421, 1437, 1438,
 1440, 1443, 1445, 1448, 1449, 1450,
 1455, 1460, 1465, 1466, 1467, 1468,
 1473, 1478, 1483, 1796, 1797, 2046,
 2053, 8373, 8400, 10324, 10575,
 12262, 12274, 18314, 18320, 18324,
 18337, 18340, 18343, 18354, 18365,
 18367, 18377, 18379, 18387, 20143,
 20145, 20155, 20434, 20550, 21595,
 21600, 21616, 21623, 21629, 21638,
 22066, 22086, 22091, 22096, 22102,
 22158, 22164, 22180, 22185, 22190,
 22196, 24015, 32981, 32982, 32983,
 32984, 32985, 32986, 36413, 36419,
 36492, 36545, 36557, 36570, 36590,
 36733, 36745, 36757, 36770, 36786,
 36801, 42028, 42052, 42086, 42502
 \tex_globaldefs:D 240
 \tex_glueexpr:D 494,
 22084, 22086, 22094, 22096, 22100,
 22102, 22117, 22124, 22129, 22132,
 30236, 42568, 42611, 42733, 42735
 \tex_glueshrink:D 495
 \tex_glueshrinkorder:D 496
 \tex_gluestretch:D ... 497, 3941, 3947
 \tex_gluestretchorder:D 498
 \tex_gluetomu:D 499
 \tex_glyphdimensionsmode:D 834
 \tex_gtoksapp:D 835
 \tex_gtokspre:D 836
 \tex_halign:D 241
 \tex_hangafter:D 242
 \tex_hangindent:D 243
 \tex_hbadness:D 244
 \tex_hbox:D 245, 36537, 36540,
 36545, 36552, 36557, 36564, 36570,
 36584, 36590, 36598, 36603, 38349
 \tex_hfi:D 1144
 \tex_hfil:D 246
 \tex_hfill:D 247
 \tex_hfilneg:D 248
 \tex_hfuzz:D 249
 \tex_hjcode:D 827
 \tex_hoffset:D 250, 1325
 \tex_holdinginserts:D 251
 \tex_hpack:D 828
 \tex_hrulerule:D . 252, 37161, 40661, 40671
 \tex_hsize:D
 253, 37480, 37505, 37506, 37556
 \tex_hskip:D 254, 22127, 22129
 \tex_hss:D 255, 36607, 36609, 36611,
 37146, 37155, 40657, 40668, 40673
 \tex_ht:D 256, 36426, 36663
 \tex_hyphen:D 149, 1229
 \tex_hyphenation:D 257
 \tex_hyphenationbounds:D 829
 \tex_hyphenationmin:D 830
 \tex_hyphenchar:D 258, 24008

- `\tex_hyphenpenalty:D` 259
- `\tex_hyphenpenaltymode:D` 831
- `\tex_if:D` 260, 1392, 1393
- `\tex_ifabsdim:D` 623, 936
- `\tex_ifabsnum:D`
..... 1055, 624, 937, 24074, 24078
- `\tex_ifcase:D` 261, 18203
- `\tex_ifcat:D` 262, 1394
- `\tex_ifcondition:D` 837
- `\tex_ifcsname:D` 500, 1402
- `\tex_ifdbbox:D` 1145
- `\tex_ifddir:D` 1146
- `\tex_ifdefined:D`
..... 501, 691, 1222, 1226, 1232,
1263, 1266, 1273, 1274, 1305, 1308,
1311, 1324, 1330, 1401, 1439, 1442
- `\tex_ifdim:D` 263, 21580
- `\tex_ifeof:D` 264, 10393
- `\tex_iffalse:D` 265, 1387
- `\tex_iffontchar:D` 502
- `\tex_ifhbox:D` 266, 36474
- `\tex_ifhmode:D` 267, 1398
- `\tex_ifincsnam:D` 671
- `\tex_ifinner:D` 268, 1400
- `\tex_ifjfont:D` 1147
- `\tex_ifmbox:D` 1148
- `\tex_ifmdir:D` 1149
- `\tex_ifmmode:D` 269, 1397
- `\tex_ifnum:D` 270, 1272, 1419
- `\tex_ifodd:D`
..... 271, 1396, 8366, 18202, 36666
- `\tex_ifprimitive:D` 625, 774
- `\tex_iftbox:D` 1150
- `\tex_iftdir:D` 1152
- `\tex_iftfont:D` 1151
- `\tex_iftrue:D` 272, 1386
- `\tex_ifvbox:D` 273, 36475
- `\tex_ifvmode:D` 274, 1399
- `\tex_ifvoid:D` 275, 36476
- `\tex_ifx:D` 276, 1238,
1255, 1282, 1288, 1294, 1300, 1395
- `\tex_ifybox:D` 1153
- `\tex_ifydir:D` 1154
- `\tex_ignoredimen:D` 642
- `\tex_ignoreligaturesinfont:D` .. 938
- `\tex_ignoreprimitiveerror:D` .. 1214
- `\tex_ignorespaces:D` 277
- `\tex_immediate:D`
..... 278, 10577, 10601, 10660
- `\tex_immediateassigned:D` 838
- `\tex_immediateassignment:D` 839
- `\tex_indent:D` 279, 2441
- `\tex_inhibitglue:D` 1155
- `\tex_inhibitxspcode:D` 1156
- `\tex_initcatcodetable:D` .. 840, 31527
- `\tex_input:D` 280,
1224, 1314, 8985, 11159, 11524, 11569
- `\tex_inputlineno:D` ... 281, 1972, 9380
- `\tex_insert:D` 282
- `\tex_insertht:D` 643, 939
- `\tex_insertpenalties:D` 283
- `\tex_interactionmode:D` 503,
2317, 2322, 2323, 36510, 36513, 36515
- `\tex_interlinepenalties:D` 504
- `\tex_interlinepenalty:D` 284
- `\tex_italiccorrection:D`
..... 148, 1230, 1326
- `\tex_jcharwidowpenalty:D` 1157
- `\tex_jfam:D` 1158
- `\tex_jfont:D` 1159
- `\tex_jis:D` 1160
- `\tex_jobname:D`
..... 285, 9070, 9235, 11017, 11018
- `\tex_kanjiskip:D` 1161, 8738
- `\tex_kansuji:D` 1162
- `\tex_kansujichar:D` 1163
- `\tex_kcatcode:D` 1164
- `\tex_kchar:D` 1203
- `\tex_kchardef:D` 1204
- `\tex_kern:D` 286, 22054
- `\tex_knaccode:D` 672
- `\tex_knbccode:D` 673
- `\tex_knbscode:D` 674
- `\tex_kuten:D` 1165
- `\tex_language:D` 287, 1315
- `\tex_lastbox:D` 288, 36490, 36492
- `\tex_lastkern:D` 289
- `\tex_lastlinedepth:D` 644
- `\tex_lastlinefit:D` 505
- `\tex_lastmatch:D` 645
- `\tex_lastnamedcs:D`
..... 841, 1836, 1848, 1878, 1892, 1918, 1928
- `\tex_lastnodechar:D` 1166
- `\tex_lastnodefont:D` 1167
- `\tex_lastnodesubtype:D` 1168
- `\tex_lastnodetype:D` 506
- `\tex_lastpenalty:D` 290
- `\tex_lastskip:D` 291
- `\tex_lastxpos:D` 646, 946
- `\tex_lastypos:D` 647, 947
- `\tex_latelua:D` 842, 11834
- `\tex_lateluafunction:D` 843
- `\tex_lccode:D` ... 292, 3672, 3682,
3692, 3704, 3775, 3777, 3780, 3810,
4196, 7248, 7300, 9210, 9222, 14013,
14014, 14036, 14037, 19945, 19947
- `\tex_leaders:D` 293
- `\tex_left:D` 294, 1331

- `\tex_leftghost:D` 844
- `\tex_lefthyphenmin:D` 295
- `\tex_leftmarginkern:D` 675
- `\tex_leftskip:D` 296
- `\tex_leqno:D` 297
- `\tex_let:D` 1552,
 - 141, 144, 298, 694, 1223, 1224,
 - 1227, 1228, 1229, 1230, 1231, 1233,
 - 1256, 1262, 1264, 1268, 1276, 1277,
 - 1284, 1290, 1296, 1302, 1306, 1309,
 - 1312, 1313, 1314, 1315, 1316, 1317,
 - 1318, 1319, 1320, 1321, 1322, 1325,
 - 1326, 1327, 1328, 1331, 1332, 1333,
 - 1386, 1387, 1388, 1389, 1390, 1391,
 - 1392, 1393, 1394, 1395, 1396, 1397,
 - 1398, 1399, 1400, 1401, 1402, 1403,
 - 1404, 1405, 1406, 1407, 1408, 1410,
 - 1411, 1412, 1413, 1414, 1415, 1416,
 - 1417, 1419, 1420, 1421, 1437, 1448,
 - 1449, 1450, 1455, 1460, 1465, 1466,
 - 1467, 1468, 1473, 1478, 1483, 2042,
 - 3673, 3683, 3694, 3706, 4156, 4225,
 - 12260, 12262, 12272, 12274, 13412,
 - 20143, 20145, 20155, 41379, 41382
- `\tex_letcharcode:D` 845
- `\tex_letterspacefont:D` 676
- `\tex_limits:D` 299, 39299
- `\tex_linedir:D` 846
- `\tex_linedirection:D` 847
- `\tex_lineendmode:D` 1175
- `\tex_linepenalty:D` 300
- `\tex_lineskip:D` 301
- `\tex_lineskiplimit:D` 302
- `\tex_localbrokenpenalty:D` 848
- `\tex_localinterlinepenalty:D` .. 849
- `\tex_localleftbox:D` 854
- `\tex_localrightbox:D` 855
- `\tex_long:D` 303, 687, 688,
 - 689, 1422, 1424, 1427, 1451, 1452,
 - 1453, 1454, 1456, 1458, 1461, 1462,
 - 1463, 1464, 1470, 1472, 1480, 1482
- `\tex_looseness:D` 304
- `\tex_lower:D` 305, 36473
- `\tex_lowercase:D` 935, 936,
 - 306, 3673, 3683, 3693, 3705, 3776,
 - 4197, 7249, 7301, 9211, 9223, 9449,
 - 14015, 14038, 19976, 20060, 20087
- `\tex_lpcode:D` 677
- `\tex_luabytecode:D` 850
- `\tex_luabytecodecall:D` 851
- `\tex_luacopyinputnodes:D` 852
- `\tex_luadef:D` 853
- `\tex_luaescapestring:D` ... 856, 11832
- `\tex_luafunction:D` 857
- `\tex_luafunctioncall:D` 858
- `\tex luatexbanner:D` 859
- `\tex luatexrevision:D` 860, 8837
- `\tex luatexversion:D`
 - 861, 1274, 1439, 3634, 3635, 8736,
 - 8833, 8835, 10514, 14323, 18331, 19073
- `\tex_mag:D` 307, 41143
- `\tex_mark:D` 308
- `\tex_marks:D` 507
- `\tex_match:D` 648
- `\tex_mathaccent:D` 309
- `\tex_mathbin:D` 310
- `\tex_mathchar:D` 311
- `\tex_mathchardef:D` 393,
 - 312, 1445, 18339, 32982, 32984, 32986
- `\tex_mathchoice:D` 313
- `\tex_mathclose:D` 314
- `\tex_mathcode:D` ... 315, 19939, 19941
- `\tex_mathdefaultsmode:D` 862
- `\tex_mathdelimitersmode:D` 863
- `\tex_mathdir:D` 864
- `\tex_mathdirection:D` 865
- `\tex_mathdisplayskipmode:D` 866
- `\tex_matheptydisplaymode:D` ... 869
- `\tex_matheqdirmode:D` 867
- `\tex_matheqnogapstep:D` 868
- `\tex_mathflattenmode:D` 870
- `\tex_mathinner:D` 316
- `\tex_mathitalicsmode:D` 871
- `\tex_mathnolimitsmode:D` 872
- `\tex_mathop:D` 317, 1316
- `\tex_mathopen:D` 318
- `\tex_mathoption:D` 873
- `\tex_mathord:D` 319
- `\tex_mathpenaltiesmode:D` 874
- `\tex_mathpunct:D` 320
- `\tex_mathrel:D` 321
- `\tex_mathrulesfam:D` 875
- `\tex_mathrulesmode:D` 877
- `\tex_mathrulethicknessmode:D` .. 879
- `\tex_mathscriptboxmode:D` 881
- `\tex_mathscriptcharmode:D` 882
- `\tex_mathscriptsmode:D` 880
- `\tex_mathstyle:D` 883
- `\tex_mathsurround:D` 322
- `\tex_mathsurroundmode:D` 884
- `\tex_mathsurroundskip:D` 885
- `\tex_maxdeadcycles:D` 323
- `\tex_maxdepth:D` 324
- `\tex_mdffivesum:D`
 - 700, 773, 1364, 11320, 14414, 14416
- `\tex_meaning:D` .. 325, 1237, 1254,
 - 1280, 1286, 1292, 1298, 1410, 1411
- `\tex_medmuskip:D` 326

<code>\tex_message:D</code>	327	<code>\tex_outputpenalty:D</code>	353
<code>\tex_middle:D</code>	508, 1332	<code>\tex_over:D</code>	354, 1319
<code>\tex_mkern:D</code>	328	<code>\tex_overfullrule:D</code>	355
<code>\tex_month:D</code> ...	329, 1292, 1296, 1317	<code>\tex_overline:D</code>	356
<code>\tex_moveleft:D</code>	330, 36467	<code>\tex_overwithdelims:D</code>	357
<code>\tex_moveright:D</code>	331, 36469	<code>\tex_pagebottomoffset:D</code>	892
<code>\tex_mskip:D</code>	332	<code>\tex_pagedepth:D</code>	358
<code>\tex_muexpr:D</code>		<code>\tex_pagedir:D</code>	893
. 509, 22178, 22180, 22188, 22190,		<code>\tex_pagedirection:D</code>	894
22194, 22196, 22200, 42557, 42616		<code>\tex_pagediscards:D</code>	512
<code>\tex_multiply:D</code>	333	<code>\tex_pagefillllstretch:D</code>	359
<code>\tex_muskip:D</code>	334	<code>\tex_pagefillstretch:D</code>	360
<code>\tex_muskipdef:D</code>	335	<code>\tex_pagefilstretch:D</code>	361
<code>\tex_mutoglua:D</code>		<code>\tex_pagefistretch:D</code>	1171
..... 382, 1567, 510, 42557, 42616		<code>\tex_pagegoal:D</code>	362
<code>\tex_newlinechar:D</code>		<code>\tex_pageheight:D</code>	651, 950
... 336, 1956, 9455, 9661, 10659,		<code>\tex_pageleftoffset:D</code>	895
12489, 12515, 12519, 12643, 13538		<code>\tex_pagerightoffset:D</code>	896
<code>\tex_noalign:D</code>	337	<code>\tex_pageshrink:D</code>	363
<code>\tex_noautospace:D</code>	1169	<code>\tex_pagestretch:D</code>	364
<code>\tex_noautoxspacing:D</code>	1170	<code>\tex_pagetopoffset:D</code>	897
<code>\tex_noboundary:D</code>	338	<code>\tex_pagetotal:D</code>	365
<code>\tex_noexpand:D</code>	339, 1406	<code>\tex_pagewidth:D</code>	652, 951
<code>\tex_nohrule:D</code>	886	<code>\tex_par:D</code>	366
<code>\tex_noindent:D</code>	340	<code>\tex_pardir:D</code>	898
<code>\tex_nokerns:D</code>	887	<code>\tex_pardirection:D</code>	899
<code>\tex_noligatures:D</code>	649	<code>\tex_parfillskip:D</code>	367
<code>\tex_noligs:D</code>	888	<code>\tex_parindent:D</code>	368
<code>\tex_nolimits:D</code>	341, 39300	<code>\tex_parshape:D</code>	369
<code>\tex_nonscript:D</code>	342	<code>\tex_parshapedimen:D</code>	513
<code>\tex_nonstopmode:D</code>	343	<code>\tex_parshapeindent:D</code>	514
<code>\tex_normaldeviate:D</code>	650, 948	<code>\tex_parshapelength:D</code>	515
<code>\tex_nospaces:D</code>	889	<code>\tex_parskip:D</code>	370
<code>\tex_novrule:D</code>	890	<code>\tex_partokencontext:D</code>	1215
<code>\tex_nulldelimiterspace:D</code>	344	<code>\tex_partokenname:D</code>	1216
<code>\tex_nullfont:D</code>	345, 20463	<code>\tex_patterns:D</code>	371
<code>\tex_number:D</code>	346, 18199, 37383	<code>\tex_pausing:D</code>	372
<code>\tex_numexpr:D</code>	384,	<code>\tex_pdfannot:D</code>	538
511, 4357, 17620, 18200, 20074, 24213		<code>\tex_pdfcatalog:D</code>	539
<code>\tex_odelcode:D</code>	1207	<code>\tex_pdfcolorstack:D</code>	541
<code>\tex_odelimiter:D</code>	1208	<code>\tex_pdfcolorstackinit:D</code>	542
<code>\tex_omathaccent:D</code>	1209	<code>\tex_pdfcompresslevel:D</code>	540
<code>\tex_omathchar:D</code>	1210	<code>\tex_pdfdecimaldigits:D</code>	543
<code>\tex_omathchardef:D</code>		<code>\tex_pdfdest:D</code>	544
1211, 1442, 1443, 18332, 18334, 18335		<code>\tex_pdfdestmargin:D</code>	545
<code>\tex_omathcode:D</code>	1212	<code>\tex_pdfendlink:D</code>	546
<code>\tex_omit:D</code>	347	<code>\tex_pdfendthread:D</code>	547
<code>\tex_openin:D</code>	348, 10326	<code>\tex_pdfextension:D</code>	900
<code>\tex_openout:D</code>	349, 10577	<code>\tex_pdffakespace:D</code>	548
<code>\tex_or:D</code>	350, 1388	<code>\tex_pdffeedback:D</code>	901
<code>\tex_oradical:D</code>	1213	<code>\tex_pdffontattr:D</code>	549
<code>\tex_outer:D</code>	351, 1318, 41415	<code>\tex_pdffontname:D</code>	550
<code>\tex_output:D</code>	352	<code>\tex_pdffontobjnum:D</code>	551
<code>\tex_outputbox:D</code>	891	<code>\tex_pdfgamma:D</code>	552

<code>\tex_pdfgentounicode:D</code>	553	<code>\tex_pdfsuppresswarningdupmap:D</code>	610
<code>\tex_pdfglyptounicode:D</code>	554	<code>\tex_pdfsuppresswarningpagegroup:D</code>	
<code>\tex_pdfhorigin:D</code>	555		612
<code>\tex_pdfimageapplygamma:D</code>	556	<code>\tex_pdftexbanner:D</code>	667
<code>\tex_pdfimagegamma:D</code>	557	<code>\tex_pdftexrevision:D</code>	668, 8816
<code>\tex_pdfimagehicolor:D</code>	558	<code>\tex_pdftexversion:D</code>	
<code>\tex_pdfimageresolution:D</code>	559		669, 1273, 8737, 8812, 8814, 14332
<code>\tex_pdfincludechars:D</code>	560	<code>\tex_pdfthread:D</code>	613
<code>\tex_pdfinclusioncopyfonts:D</code>	561	<code>\tex_pdfthreadmargin:D</code>	614
<code>\tex_pdfinclusionerrorlevel:D</code>	563	<code>\tex_pdftrailer:D</code>	615
<code>\tex_pdfinfo:D</code>	564	<code>\tex_pdftrailerid:D</code>	616
<code>\tex_pdfinfoomitdate:D</code>	565	<code>\tex_pdfuniqueresname:D</code>	617
<code>\tex_pdfinterwordspaceoff:D</code>	566	<code>\tex_pdfvariable:D</code>	902
<code>\tex_pdfinterwordspaceon:D</code>	567	<code>\tex_pdfvorigin:D</code>	618
<code>\tex_pdflastannot:D</code>	568	<code>\tex_pdfxform:D</code>	619, 958
<code>\tex_pdflastlink:D</code>	569	<code>\tex_pdfxformname:D</code>	620
<code>\tex_pdflastobj:D</code>	570	<code>\tex_pdfximage:D</code>	621, 959
<code>\tex_pdflastxform:D</code>	571, 941	<code>\tex_pdfximagebbox:D</code>	622
<code>\tex_pdflastximage:D</code>	572, 943	<code>\tex_penalty:D</code>	373
<code>\tex_pdflastximagecolordepth:D</code>	574	<code>\tex_pkmode:D</code>	653
<code>\tex_pdflastximagepages:D</code>	575, 945	<code>\tex_pkresolution:D</code>	654
<code>\tex_pdflinkmargin:D</code>	576	<code>\tex_postbreakpenalty:D</code>	1172
<code>\tex_pdfliteral:D</code>	577	<code>\tex_postdisplaypenalty:D</code>	374
<code>\tex_pdfmajorversion:D</code>	580	<code>\tex_posttexhyphenchar:D</code>	903
<code>\tex_pdfmapfile:D</code>	578, 1276	<code>\tex_postthyphenchar:D</code>	904
<code>\tex_pdfmapline:D</code>	579, 1277	<code>\tex_prebinoppenalty:D</code>	905
<code>\tex_pdfminorversion:D</code>	581	<code>\tex_prebreakpenalty:D</code>	1173
<code>\tex_pdfnames:D</code>	582	<code>\tex_predisplaydirection:D</code>	516
<code>\tex_pdfnobuiltintounicode:D</code>	583	<code>\tex_predisplaygapfactor:D</code>	906
<code>\tex_pdfobj:D</code>	584	<code>\tex_predisplaypenalty:D</code>	375
<code>\tex_pdfobjcompresslevel:D</code>	585	<code>\tex_predisplaysize:D</code>	376
<code>\tex_pdfomitcharset:D</code>	586	<code>\tex_preexhyphenchar:D</code>	907
<code>\tex_pdfoutline:D</code>	587	<code>\tex_prehyphenchar:D</code>	908
<code>\tex_pdfoutput:D</code>	618, 588, 949, 1309, 8783, 8789, 8797, 9250	<code>\tex_prependkern:D</code>	656
<code>\tex_pdfpageattr:D</code>	589	<code>\tex_prerelpenalty:D</code>	909
<code>\tex_pdfpagebox:D</code>	590	<code>\tex_pretolerance:D</code>	377
<code>\tex_pdfpageref:D</code>	591	<code>\tex_prevdepth:D</code>	378
<code>\tex_pdfpageresources:D</code>	592	<code>\tex_prevgraf:D</code>	379
<code>\tex_pdfpagesattr:D</code>	593	<code>\tex_primitive:D</code>	655, 775, 9079, 9089
<code>\tex_pdfptexuseunderscore:D</code>	594	<code>\tex_protected:D</code>	
<code>\tex_pdfrefobj:D</code>	595		517, 1451, 1453, 1456, 1457, 1458, 1459, 1461, 1462, 1463, 1464, 1475, 1477, 1480, 1482, 41415
<code>\tex_pdfrefxform:D</code>	596, 955	<code>\tex_protrudechars:D</code>	657, 778, 952
<code>\tex_pdfrefximage:D</code>	597, 956	<code>\tex_protrusionboundary:D</code>	910
<code>\tex_pdfrestore:D</code>	598	<code>\tex_ptexfontname:D</code>	1174
<code>\tex_pdfretval:D</code>	599	<code>\tex_ptexminorversion:D</code>	
<code>\tex_pdfrunninglinkoff:D</code>	600		1176, 8825, 8846
<code>\tex_pdfrunninglinkon:D</code>	601	<code>\tex_ptexrevision:D</code>	1177, 8826, 8847
<code>\tex_pdfsave:D</code>	602	<code>\tex_ptextracingfonts:D</code>	1178
<code>\tex_pdfsetmatrix:D</code>	603	<code>\tex_ptexversion:D</code>	
<code>\tex_pdfstartlink:D</code>	604		1179, 8820, 8823, 8841, 8844
<code>\tex_pdfstarttthread:D</code>	605	<code>\tex_pxdimen:D</code>	658, 953
<code>\tex_pdfsuppressptexinfo:D</code>	606	<code>\tex_quitvmode:D</code>	678
<code>\tex_pdfsuppresswarningdupdest:D</code>	608		

- \tex_radical:D 380
- \tex_raise:D 381, 36471
- \tex_randomseed:D 659, 954, 9114
- \tex_read:D 382, 9566, 10413
- \tex_readline:D 518, 10430
- \tex_readpapersizespecial:D .. 1180
- \tex_relax:D 382,
1066, 1567, 383, 1415, 18201, 21582
- \tex_relpentalty:D 384
- \tex_resettimer:D 660, 776
- \tex_right:D 385, 1333
- \tex_rightghost:D 911
- \tex_righthypenmin:D 386
- \tex_rightmarginkern:D 679
- \tex_rightskip:D 387
- \tex_romannumeral:D 405,
433, 434, 1567, 388, 1408, 1420,
1801, 19988, 24215, 31047, 31064,
31066, 31610, 32092, 32149, 32195,
32325, 32329, 32629, 32638, 41903
- \tex_rrcode:D 680
- \tex_savecatcodetable:D
..... 912, 31565, 31627
- \tex_savepos:D 661, 957
- \tex_savinghyphcodes:D 519
- \tex_savingvdiscards:D 520
- \tex_scantextokens:D
..... 913, 12568, 12595, 12629
- \tex_scantokens:D 521, 12501, 12562,
12646, 42026, 42050, 42084, 42500
- \tex_scriptbaselineshiftfactor:D
..... 1182
- \tex_scriptfont:D 389
- \tex_scriptscriptbaselineshiftfactor:D
..... 1184
- \tex_scriptscriptfont:D 390
- \tex_scriptscriptstyle:D 391
- \tex_scriptspace:D 392
- \tex_scriptstyle:D 393
- \tex_scrollmode:D 394
- \tex_setbox:D .. 395, 36411, 36413,
36417, 36419, 36437, 36446, 36455,
36490, 36492, 36540, 36545, 36552,
36557, 36564, 36570, 36584, 36590,
36641, 36702, 36728, 36733, 36740,
36745, 36752, 36757, 36764, 36770,
36780, 36786, 36797, 36801, 38349
- \tex_setfontid:D 914
- \tex_setlanguage:D 396
- \tex_setrandomseed:D . 662, 960, 9119
- \tex_sfcode:D 397, 19957, 19959
- \tex_shapemode:D 915
- \tex_shbscode:D 681
- \tex_shellescape:D ... 663, 777, 9147
- \tex_shipout:D 398, 1233, 1257
- \tex_show:D 399
- \tex_showbox:D 400, 36527
- \tex_showboxbreadth:D ... 401, 36523
- \tex_showboxdepth:D 402, 36524
- \tex_showgroups:D 522, 2321
- \tex_showifs:D 523
- \tex_showlists:D 403
- \tex_showmode:D 1185
- \tex_showstream:D
... 1217, 10667, 10668, 10672, 10673
- \tex_showthe:D 404
- \tex_showtokens:D
..... 751, 524, 1328, 9665, 13542
- \tex_sjis:D 1186
- \tex_skewchar:D 405
- \tex_skip:D
406, 3813, 3842, 3861, 3925, 3941, 3947
- \tex_skipdef:D 407
- \tex_space:D 147
- \tex_spacefactor:D 408
- \tex_spaceskip:D 409
- \tex_span:D 410
- \tex_special:D 411
- \tex_splitbotmark:D 412
- \tex_splitbotmarks:D 525
- \tex_splitdiscards:D 526
- \tex_splitfirstmark:D 413
- \tex_splitfirstmarks:D 527
- \tex_splitmaxdepth:D 414
- \tex_splittopskip:D 415
- \tex_stbscode:D 682
- \tex_strcmp:D ... 698, 1337, 11441,
13816, 20341, 20357, 20363, 24591
- \tex_string:D 416, 1236,
1240, 1281, 1287, 1293, 1299, 1413
- \tex_suppressfontnotfounderror:D 704
- \tex_suppressifcsnameerror:D .. 916
- \tex_suppresslongerror:D 917
- \tex_suppressmathparerror:D ... 918
- \tex_suppressoutererror:D 919
- \tex_suppressprimitiveerror:D .. 921
- \tex_synctex:D 683
- \tex_tabskip:D 417
- \tex_tagcode:D 684
- \tex_tate:D 1187
- \tex_tbaselineshift:D 1188
- \tex_textbaselineshiftfactor:D 1190
- \tex_textdir:D 922
- \tex_textdirection:D 923
- \tex_textfont:D 418
- \tex_textstyle:D 419
- \tex_TeXTeXtstate:D 528
- \tex_tfont:D 1191

<code>\tex_the:D</code>	965
. 382, 427, 1102, 1108, 1109, 121,	
420, 1972, 2290, 2493, 2497, 3304,	
3336, 3385, 3386, 3417, 3418, 3424,	
3425, 3940, 4061, 4361, 4380, 4386,	
4392, 4400, 6329, 6331, 6345, 6346,	
6348, 6349, 6581, 6624, 6846, 6945,	
9114, 17735, 18212, 18213, 18414,	
18415, 19871, 19941, 19947, 19953,	
19959, 21837, 21838, 21839, 21909,	
21910, 25230, 25719, 36510, 41892	
<code>\tex_thickmuskip:D</code>	421
<code>\tex_thinmuskip:D</code>	422
<code>\tex_time:D</code>	423, 1280, 1284
<code>\tex_tojis:D</code>	1192
<code>\tex_toks:D</code>	
.... 424, 3277, 3304, 3336, 3374,	
3385, 3386, 3417, 3418, 3424, 3425,	
3429, 3439, 3449, 3758, 3776, 3940,	
4361, 4363, 4364, 4366, 4371, 4380,	
4386, 4392, 17735, 17747, 17748, 17749	
<code>\tex_toksapp:D</code>	924, 4388, 4389
<code>\tex_toksdef:D</code>	425, 3556
<code>\tex_tokspre:D</code>	925
<code>\tex_tolerance:D</code>	426
<code>\tex_topmark:D</code>	427
<code>\tex_topmarks:D</code>	529
<code>\tex_topskip:D</code>	428
<code>\tex_toucs:D</code>	1193
<code>\tex_tpack:D</code>	926
<code>\tex_tracingassigns:D</code>	530
<code>\tex_tracingcommands:D</code>	429
<code>\tex_tracingfonts:D</code>	
.... 664, 961, 1262, 1264, 1268	
<code>\tex_tracinggroups:D</code>	531
<code>\tex_tracingifs:D</code>	532
<code>\tex_tracinglostchars:D</code>	430
<code>\tex_tracingmacros:D</code>	431
<code>\tex_tracingnesting:D</code>	
.... 533, 8979, 11152, 12486	
<code>\tex_tracingonline:D</code>	
.... 432, 2318, 2324, 2325, 36525	
<code>\tex_tracingoutput:D</code>	433
<code>\tex_tracingpages:D</code>	434
<code>\tex_tracingparagraphs:D</code>	435
<code>\tex_tracingrestores:D</code>	436
<code>\tex_tracingscantokens:D</code>	534
<code>\tex_tracingstacklevels:D</code>	1218
<code>\tex_tracingstats:D</code>	437
<code>\tex_uccode:D</code>	438, 19951, 19953
<code>\tex_Uchar:D</code>	963
<code>\tex_Ucharcat:D</code> 964, 1349, 20025, 20030	
<code>\tex_uchyph:D</code>	439
<code>\tex_ucs:D</code>	1194
<code>\tex_Udelcode:D</code>	965
<code>\tex_Udelcodenum:D</code>	966
<code>\tex_Udelimiter:D</code>	967
<code>\tex_Udelimiterover:D</code>	968
<code>\tex_Udelimiterunder:D</code>	969
<code>\tex_Uhextensible:D</code>	970
<code>\tex_Uleft:D</code>	971
<code>\tex_Umathaccent:D</code>	972
<code>\tex_Umathaxis:D</code>	973
<code>\tex_Umathbinbinspacing:D</code>	974
<code>\tex_Umathbinclosespacing:D</code> ...	975
<code>\tex_Umathbininnerspacing:D</code> ...	976
<code>\tex_Umathbinopenspacing:D</code>	977
<code>\tex_Umathbinopspacing:D</code>	978
<code>\tex_Umathbinordspacing:D</code>	979
<code>\tex_Umathbinpunctspacing:D</code> ...	980
<code>\tex_Umathbinrelspacing:D</code>	981
<code>\tex_Umathchar:D</code>	982
<code>\tex_Umathcharclass:D</code>	983
<code>\tex_Umathchardef:D</code>	984
<code>\tex_Umathcharfam:D</code>	985
<code>\tex_Umathcharnum:D</code>	986
<code>\tex_Umathcharnumdef:D</code>	987
<code>\tex_Umathcharslot:D</code>	988
<code>\tex_Umathclosebinspacing:D</code> ...	989
<code>\tex_Umathcloseclosespacing:D</code> ..	991
<code>\tex_Umathcloseinnerspacing:D</code> ..	993
<code>\tex_Umathcloseopenspacing:D</code> ...	994
<code>\tex_Umathcloseopspacing:D</code>	995
<code>\tex_Umathcloseordspacing:D</code> ...	996
<code>\tex_Umathclosepunctspacing:D</code> ..	998
<code>\tex_Umathcloserelspacing:D</code> ...	999
<code>\tex_Umathcode:D</code>	1000, 8753
<code>\tex_Umathcodenum:D</code>	1001
<code>\tex_Umathconnectoroverlapmin:D</code>	1003
<code>\tex_Umathfractiondelsize:D</code> ..	1004
<code>\tex_Umathfractiondenomdown:D</code> .	1006
<code>\tex_Umathfractiondenomvgap:D</code> .	1008
<code>\tex_Umathfractionnumup:D</code>	1009
<code>\tex_Umathfractionnumvgap:D</code> ...	1010
<code>\tex_Umathfractionrule:D</code>	1011
<code>\tex_Umathinnerbinspacing:D</code> ...	1012
<code>\tex_Umathinnerclosespacing:D</code> .	1014
<code>\tex_Umathinnerinnerspacing:D</code> .	1016
<code>\tex_Umathinneropenspacing:D</code> .	1017
<code>\tex_Umathinneropspacing:D</code> ...	1018
<code>\tex_Umathinnerordspacing:D</code> ...	1019
<code>\tex_Umathinnerpunctspacing:D</code> .	1021
<code>\tex_Umathinnerrelspacing:D</code> ...	1022
<code>\tex_Umathlimitabovebgap:D</code> ...	1023
<code>\tex_Umathlimitabovekern:D</code> ...	1024
<code>\tex_Umathlimitabovevgap:D</code> ...	1025
<code>\tex_Umathlimitbelowbgap:D</code> ...	1026
<code>\tex_Umathlimitbelowkern:D</code> ...	1027

<code>\tex_Umathlimitbelowvgap:D</code> . . .	1028	<code>\tex_Umathrelpunctspacing:D</code> . .	1090
<code>\tex_Umathnolimitsubfactor:D</code> .	1029	<code>\tex_Umathrelrelspacing:D</code>	1091
<code>\tex_Umathnolimitsupfactor:D</code> .	1030	<code>\tex_Umathskewedfractionhgap:D</code>	1093
<code>\tex_Umathopbinspacing:D</code>	1031	<code>\tex_Umathskewedfractionvgap:D</code>	1095
<code>\tex_Umathopclorespacing:D</code> . . .	1032	<code>\tex_Umathspaceafterscript:D</code> .	1096
<code>\tex_Umathopenbinspacing:D</code> . . .	1033	<code>\tex_Umathstackdenomdown:D</code> . . .	1097
<code>\tex_Umathopenclorespacing:D</code> .	1034	<code>\tex_Umathstacknumup:D</code>	1098
<code>\tex_Umathopeninnerspacing:D</code> .	1035	<code>\tex_Umathstackvgap:D</code>	1099
<code>\tex_Umathopenopenspacing:D</code> . .	1036	<code>\tex_Umathsubshiftdown:D</code>	1100
<code>\tex_Umathopenopspacing:D</code>	1037	<code>\tex_Umathsubshiftdrop:D</code>	1101
<code>\tex_Umathopenordspacing:D</code> . . .	1038	<code>\tex_Umathsubsupshiftdown:D</code> . .	1102
<code>\tex_Umathopenpunctspacing:D</code> .	1039	<code>\tex_Umathsubsupvgap:D</code>	1103
<code>\tex_Umathopenrelspacing:D</code> . . .	1040	<code>\tex_Umathsubtopmax:D</code>	1104
<code>\tex_Umathoperatorsiz:D</code>	1041	<code>\tex_Umathsupbottommin:D</code>	1105
<code>\tex_Umathopinnerspacing:D</code> . . .	1042	<code>\tex_Umathsupshiftdrop:D</code>	1106
<code>\tex_Umathopopenspacing:D</code>	1043	<code>\tex_Umathsupshiftup:D</code>	1107
<code>\tex_Umathopopspacing:D</code>	1044	<code>\tex_Umathsupsubbottommax:D</code> . .	1108
<code>\tex_Umathopordspacing:D</code>	1045	<code>\tex_Umathunderbarkern:D</code>	1109
<code>\tex_Umathoppunctspacing:D</code> . . .	1046	<code>\tex_Umathunderbarrule:D</code>	1110
<code>\tex_Umathoprelspacing:D</code>	1047	<code>\tex_Umathunderbarvgap:D</code>	1111
<code>\tex_Umathordbinspacing:D</code>	1048	<code>\tex_Umathunderdelimiterbgap:D</code>	1113
<code>\tex_Umathordclorespacing:D</code> . .	1049	<code>\tex_Umathunderdelimitervgap:D</code>	1115
<code>\tex_Umathordinnerspacing:D</code> . .	1050	<code>\tex_Umiddle:D</code>	1116
<code>\tex_Umathordopenspacing:D</code> . . .	1051	<code>\tex_undefine:D</code>	31117
<code>\tex_Umathordopspacing:D</code>	1052	<code>\tex_undefined:D</code>	
<code>\tex_Umathordordspacing:D</code>	1053	 483, 945, 1018, 1262,	
<code>\tex_Umathordpunctspacing:D</code> . .	1054	1276, 1277, 1284, 1290, 1296, 1302,	
<code>\tex_Umathordrelspacing:D</code>	1055	2059, 3227, 3673, 3683, 3693, 3694,	
<code>\tex_Umathoverbarkern:D</code>	1056	3705, 3706, 3785, 3886, 4195, 19106,	
<code>\tex_Umathoverbarrule:D</code>	1057	20859, 21159, 22506, 30877, 30894	
<code>\tex_Umathoverbarvgap:D</code>	1058	<code>\tex_underline:D</code>	440, 1231
<code>\tex_Umathoverdelimiterbgap:D</code> .	1060	<code>\tex_unescapehex:D</code>	665
<code>\tex_Umathoverdelimitervgap:D</code> .	1062	<code>\tex_unexpanded:D</code>	
<code>\tex_Umathpunctbinspacing:D</code> . .	1063	 433, 535, 1321, 1407, 2725	
<code>\tex_Umathpunctclorespacing:D</code> .	1065	<code>\tex_unhbox:D</code>	441, 36613
<code>\tex_Umathpunctinnerspacing:D</code> .	1067	<code>\tex_unhcopy:D</code>	442, 36612
<code>\tex_Umathpunctopenspacing:D</code> .	1068	<code>\tex_uniformdeviate:D</code>	
<code>\tex_Umathpunctopspacing:D</code> . . .	1069	 866, 1268, 1269, 666,	
<code>\tex_Umathpunctordspacing:D</code> . .	1070	962, 17746, 30305, 30306, 30487, 30490	
<code>\tex_Umathpunctpunctspacing:D</code> .	1072	<code>\tex_unkern:D</code>	443
<code>\tex_Umathpunctrelspacing:D</code> . .	1073	<code>\tex_unless:D</code>	536, 1391
<code>\tex_Umathquad:D</code>	1074	<code>\tex_Unosubscript:D</code>	1117
<code>\tex_Umathradicaldegreeafter:D</code>	1076	<code>\tex_Unosuperscript:D</code>	1118
<code>\tex_Umathradicaldegreebefore:D</code>	1078	<code>\tex_unpenalty:D</code>	444
<code>\tex_Umathradicaldegreeraise:D</code>	1080	<code>\tex_unskip:D</code>	445
<code>\tex_Umathradicalkern:D</code>	1081	<code>\tex_unvbox:D</code>	446, 36793
<code>\tex_Umathradicalrule:D</code>	1082	<code>\tex_unvcopy:D</code>	447, 36792
<code>\tex_Umathradicalvgap:D</code>	1083	<code>\tex_Uoverdelimiter:D</code>	1119
<code>\tex_Umathrelbinspacing:D</code>	1084	<code>\tex_uppercase:D</code>	448
<code>\tex_Umathrelclorespacing:D</code> . .	1085	<code>\tex_uptexrevision:D</code>	1205, 8851
<code>\tex_Umathrelinnerspacing:D</code> . .	1086	<code>\tex_uptexversion:D</code>	1206, 8850
<code>\tex_Umathrelopenspacing:D</code> . . .	1087	<code>\tex_Uradical:D</code>	1120
<code>\tex_Umathrelopspacing:D</code>	1088	<code>\tex_Uright:D</code>	1121
<code>\tex_Umathrelordspacing:D</code>	1089	<code>\tex_Uroot:D</code>	1122

<code>\tex_Uskewed:D</code>	1123	<code>\tex_XeTeXfindvariationbyname:D</code>	720
<code>\tex_Uskewedwithdelims:D</code>	1124	<code>\tex_XeTeXfirstfontchar:D</code>	721
<code>\tex_Ustack:D</code>	1125	<code>\tex_XeTeXfonttype:D</code>	722
<code>\tex_Ustartdisplaymath:D</code>	1126	<code>\tex_XeTeXgenerateactualtext:D</code> .	724
<code>\tex_Ustartmath:D</code>	1127	<code>\tex_XeTeXglyph:D</code>	725
<code>\tex_Ustopdisplaymath:D</code>	1128	<code>\tex_XeTeXglyphbounds:D</code>	726
<code>\tex_Ustopmath:D</code>	1129	<code>\tex_XeTeXglyphindex:D</code>	727
<code>\tex_Usubscript:D</code>	1130	<code>\tex_XeTeXglyphname:D</code>	728
<code>\tex_USuperscript:D</code>	1131	<code>\tex_XeTeXhyphenatablelength:D</code> .	767
<code>\tex_Underdelimitter:D</code>	1132	<code>\tex_XeTeXinputencoding:D</code>	729
<code>\tex_Uvextensible:D</code>	1133	<code>\tex_XeTeXinputnormalization:D</code> .	731
<code>\tex_vadjust:D</code>	449	<code>\tex_XeTeXinterchartokenstate:D</code>	733
<code>\tex_valign:D</code>	450	<code>\tex_XeTeXinterchartoks:D</code>	734
<code>\tex_variablefam:D</code>	927	<code>\tex_XeTeXinterwordspaceshaping:D</code>	765
<code>\tex_vbadness:D</code>	451	<code>\tex_XeTeXisdefaultselector:D</code> ..	736
<code>\tex_vbox:D</code>		<code>\tex_XeTeXisexclusivefeature:D</code> .	738
. 452, 36617, 36621, 36641, 36675,		<code>\tex_XeTeXlastfontchar:D</code>	739
36684, 36696, 36728, 36733, 36752,		<code>\tex_XeTeXlinebreaklocale:D</code> ...	741
36757, 36764, 36770, 36780, 36786		<code>\tex_XeTeXlinebreakpenalty:D</code> ..	742
<code>\tex_vcenter:D</code>	453, 1320	<code>\tex_XeTeXlinebreakskip:D</code>	740
<code>\tex_vfi:D</code>	1199	<code>\tex_XeTeXOTcountfeatures:D</code> ...	743
<code>\tex_vfil:D</code>	454	<code>\tex_XeTeXOTcountlanguages:D</code> ..	744
<code>\tex_vfill:D</code>	455	<code>\tex_XeTeXOTcountscripts:D</code>	745
<code>\tex_vfilneg:D</code>	456	<code>\tex_XeTeXOTfeaturetag:D</code>	746
<code>\tex_vfuzz:D</code>	457	<code>\tex_XeTeXOTlanguagetag:D</code>	747
<code>\tex_voffset:D</code>	458, 1327	<code>\tex_XeTeXOTscripttag:D</code>	748
<code>\tex_vpack:D</code>	928	<code>\tex_XeTeXpdiffile:D</code>	749
<code>\tex_vrule:D</code>		<code>\tex_XeTeXpdfpagecount:D</code>	750
... 459, 37167, 38414, 40656, 40674		<code>\tex_XeTeXpicfile:D</code>	751
<code>\tex_vsize:D</code>	460	<code>\tex_XeTeXrevision:D</code> . 752, 8859, 9085	
<code>\tex_vskip:D</code>	461, 22130, 22132	<code>\tex_XeTeXselectorcode:D</code>	763
<code>\tex_vsplit:D</code>	462, 36797, 36802	<code>\tex_XeTeXselectorname:D</code>	753
<code>\tex_vss:D</code>	463, 40663, 40670	<code>\tex_XeTeXtracingfonts:D</code>	754
<code>\tex_vtop:D</code>	464, 36619, 36623,	<code>\tex_XeTeXupwardsmode:D</code>	755
36677, 36686, 36698, 36740, 36745		<code>\tex_XeTeXuseglyphmetrics:D</code> ...	756
<code>\tex_wd:D</code>	465, 36428	<code>\tex_XeTeXvariation:D</code>	757
<code>\tex_widowpenalties:D</code>	537	<code>\tex_XeTeXvariationdefault:D</code> ..	758
<code>\tex_widowpenalty:D</code>	466	<code>\tex_XeTeXvariationmax:D</code>	759
<code>\tex_wordboundary:D</code>	929	<code>\tex_XeTeXvariationmin:D</code>	760
<code>\tex_write:D</code> . 467, 10638, 10641, 10660		<code>\tex_XeTeXvariationname:D</code>	761
<code>\tex_xdef:D</code>		<code>\tex_XeTeXversion:D</code>	
... 468, 1449, 1450, 1454, 1459, 1464		. 762, 8744, 8858, 14322, 18333, 19074	
<code>\tex_XeTeXcharclass:D</code>	705	<code>\tex_xkanjiskip:D</code>	1195
<code>\tex_XeTeXcharglyph:D</code>	706	<code>\tex_xleaders:D</code>	469
<code>\tex_XeTeXcountfeatures:D</code>	707	<code>\tex_xspaceskip:D</code>	470
<code>\tex_XeTeXcountglyphs:D</code>	708	<code>\tex_xspcode:D</code>	1196
<code>\tex_XeTeXcountselectors:D</code>	709	<code>\tex_xtoksapp:D</code>	930
<code>\tex_XeTeXcountvariations:D</code> ...	710	<code>\tex_xtokspre:D</code>	931
<code>\tex_XeTeXdashbreakstate:D</code>	712	<code>\tex_ybaselineshift:D</code>	1197
<code>\tex_XeTeXdefaultencoding:D</code> ...	711	<code>\tex_year:D</code>	471, 1298, 1302
<code>\tex_XeTeXfeaturecode:D</code>	713	<code>\tex_yoko:D</code>	1198
<code>\tex_XeTeXfeaturename:D</code>	714	text commands:	
<code>\tex_XeTeXfindfeaturebyname:D</code> ..	716	<code>\l_text_accents_tl</code>	32931, 33168
<code>\tex_XeTeXfindselectorbyname:D</code> .	718		

- \text_bcp_parse:n 309, 33392, 36080, 36080
- \l_text_case_exclude_arg_tl 303, 304, 306, 32933, 33132, 33584
- \text_case_switch:nnnn 306, 33125, 33639, 33939, 33939
- \text_declare_case_equivalent:Nn 305, 33893, 33893
- \text_declare_expand_equivalent:Nn 303, 33326, 33326, 33331, 33334, 33349
- \text_declare_lowercase_exclusion:n 305, 33931, 33931
- \text_declare_lowercase_mapping:nn 305, 33898, 33898
- \text_declare_lowercase_mapping:nnn 305, 33898, 33914
- \text_declare_purify_equivalent:Nn 306, 35712, 35712, 35717, 35725, 35726, 35727, 35728, 35729, 35731, 35732, 35734, 35737, 35738, 35744, 35746, 35747, 35748, 35752, 35785, 35800
- \text_declare_titlecase_exclusion:n 305, 33931, 33933
- \text_declare_titlecase_mapping:nn 305, 33898, 33900
- \text_declare_titlecase_mapping:nnn 305, 33898, 33916
- \text_declare_uppercase_exclusion:n 305, 33931, 33935
- \text_declare_uppercase_mapping:nn 305, 33898, 33902, 34932
- \text_declare_uppercase_mapping:nnn 305, 33898, 33918
- \text_expand:n 303, 304, 306–308, 32987, 32987, 33382, 35195, 35200, 35323, 35328, 35512
- \l_text_expand_exclude_tl 303, 306, 32944, 33131
- \l_text_letterlike_tl . 32931, 33188
- \text_lowercase:n 143, 204, 304, 33355, 33355, 41558, 41561, 41563, 41565, 41577, 41580, 41605, 41606, 41621, 41622
- \text_lowercase:nn 304, 33355, 33363, 33365, 41566, 41568
- \text_map_break: 308, 34937, 34941, 34973, 35014, 35084, 35144, 35145, 35147, 35266, 35489, 35499
- \text_map_break:n .. 308, 34937, 35146
- \text_map_function:nN 307, 35193, 35193, 35321, 35487
- \text_map_inline:nn 307, 35482, 35482
- \text_map_tokens:nn 307, 35193, 35198, 35321
- \l_text_math_arg_tl 303, 306, 32940, 33130, 33583, 35629
- \l_text_math_delims_tl 303, 306, 32942, 33046, 33507, 34976, 35558
- \text_purify:n 306, 35506, 35506, 35515
- \text_titlecase:n 41556, 41557
- \text_titlecase:nn 41556, 41560
- \text_titlecase_all:n 143, 304, 33355, 33359
- \text_titlecase_all:nn 304, 33355, 33369, 33371
- \l_text_titlecase_check_letter_bool 305, 306, 33353, 33785
- \text_titlecase_first:n 304, 33355, 33361, 41556, 41558, 41575, 41577, 41609, 41610, 41625, 41626, 41632, 41634
- \text_titlecase_first:nn 304, 33355, 33372, 33374, 41559, 41561, 41578, 41580
- \text_uppercase:n 143, 204, 304, 33355, 33357, 41569, 41571, 41607, 41608, 41623, 41624
- \text_uppercase:nn 304, 33355, 33366, 33368, 41572, 41574
- \text_words_map_function:nN 308, 35321, 35497
- \text_words_map_inline:nn 308, 35482, 35492
- \text_words_map_tokens:nn 308, 35326
- text internal commands:
 - \c_text_bcp_normal_prop 36050, 36089
 - __text_bcp_parse_auxi:n 36080, 36084, 36086
 - __text_bcp_parse_auxii:nn 36080, 36088, 36092
 - __text_bcp_parse_auxiii:n 36080, 36095, 36096, 36098
 - __text_bcp_parse_auxiv:w 36080, 36100, 36102
 - __text_bcp_parse_auxv:w 36080, 36107, 36111
 - __text_bcp_parse_auxvi:n 36080, 36120, 36132, 36154
 - __text_bcp_parse_auxvii:NNNN ... 36080, 36133, 36134
 - __text_bcp_parse_count:n 36080, 36104, 36115, 36149, 36184, 36209, 36244, 36324, 36327, 36356, 36363
 - __text_bcp_parse_count_auxi:w .. 36080, 36365, 36369

- _text_bcp_parse_count_auxii:w .
..... [36080](#), [36372](#), [36375](#)
- _text_bcp_parse_count_auxiii:w
..... [36080](#), [36376](#), [36377](#)
- _text_bcp_parse_count_auxiv:N .
..... [36080](#), [36378](#), [36379](#)
- _text_bcp_parse_ext:n
..... [36080](#), [36117](#),
[36151](#), [36186](#), [36211](#), [36278](#), [36280](#)
- _text_bcp_parse_ext:nN
..... [36080](#), [36287](#), [36289](#), [36337](#)
- _text_bcp_parse_ext:nNnw
..... [36080](#), [36302](#), [36316](#), [36325](#)
- _text_bcp_parse_extlang:n
..... [36080](#), [36119](#), [36143](#)
- _text_bcp_parse_extlang:nn ...
..... [36080](#), [36153](#), [36166](#)
- _text_bcp_parse_extlang:nw ...
..... [36080](#), [36144](#), [36145](#), [36173](#)
- _text_bcp_parse_private:nw ...
.. [36080](#), [36285](#), [36308](#), [36348](#), [36357](#)
- _text_bcp_parse_region:n
.. [36080](#), [36118](#), [36152](#), [36187](#), [36200](#)
- _text_bcp_parse_region:w
..... [36080](#), [36203](#), [36205](#)
- _text_bcp_parse_script:n
..... [36080](#), [36139](#), [36175](#)
- _text_bcp_parse_script:w
..... [36080](#), [36178](#), [36180](#)
- _text_bcp_parse_variant:n
[36080](#), [36121](#), [36122](#), [36123](#), [36124](#),
[36140](#), [36155](#), [36156](#), [36157](#), [36158](#),
[36189](#), [36190](#), [36191](#), [36192](#), [36213](#),
[36214](#), [36215](#), [36216](#), [36235](#), [36238](#)
- _text_bcp_parse_variant:nn ...
..... [36080](#), [36248](#),
[36249](#), [36250](#), [36251](#), [36259](#), [36272](#)
- _text_bcp_parse_variant:nw ...
..... [36080](#), [36239](#), [36240](#), [36260](#)
- _text_bcp_parse_variant_chk:n .
..... [36080](#), [36188](#), [36212](#), [36224](#)
- _text_bcp_parse_variant_chk:nn
..... [36080](#), [36247](#), [36261](#)
- _text_bcp_parse_variant_-
chk:NNNN [36080](#), [36225](#), [36226](#)
- _text_bcp_parse_variant_-
chk:NNNNn [36080](#), [36262](#), [36263](#)
- _text_bcp_parse_variant_end:n .
..... [36080](#)
- _text_bcp_parse_variant_end:nn
..... [36246](#), [36275](#)
- _text_case_switch_marker:
..... [33939](#), [33941](#), [33944](#)
- _text_change_case:nnn
..... [33355](#), [33356](#), [33358](#),
[33360](#), [33364](#), [33367](#), [33370](#), [33375](#)
- _text_change_case:nnnn
.. [33362](#), [33373](#), [33376](#), [33377](#), [33377](#)
- _text_change_case_auxi:nnnn ...
..... [33377](#), [33381](#), [33386](#)
- _text_change_case_auxii:nnnn ..
..... [33377](#),
[33389](#), [33400](#), [33409](#), [33415](#), [33419](#)
- _text_change_case_auxiii:nnnnn .
..... [33402](#), [33406](#), [33417](#)
- _text_change_case_bcp:nn
..... [33391](#), [33395](#)
- _text_change_case_bcp:nnn .. [33377](#)
- _text_change_case_bcp:nnnnn . [33377](#)
- _text_change_case_bcp:nnnnnnnn
..... [33377](#), [33396](#), [33397](#)
- _text_change_case_boundary_-
upper_el-x-iota:Nnnnw [34463](#)
- _text_change_case_boundary_-
upper_el:nnnN . [34463](#), [34467](#), [34473](#)
- _text_change_case_boundary_-
upper_el:nnnn . [34463](#), [34479](#), [34483](#)
- _text_change_case_boundary_-
upper_el:Nnnnw [34463](#), [34463](#), [34472](#)
- _text_change_case_boundary_-
upper_el:nnnnw [34463](#), [34492](#), [34495](#)
- _text_change_case_break:
.... [33377](#), [33442](#), [33455](#), [33456](#),
[33495](#), [33501](#), [33543](#), [33673](#), [34569](#)
- _text_change_case_breathing:nnnn
..... [34493](#), [34507](#), [34507](#)
- _text_change_case_breathing:nnnnn
..... [34507](#), [34511](#), [34520](#)
- _text_change_case_breathing:nnnnnnw
..... [34507](#), [34532](#), [34536](#), [34545](#)
- _text_change_case_breathing:nnnnnw
..... [34507](#), [34523](#), [34526](#)
- _text_change_case_breathing_-
aux:nnnN [34507](#), [34562](#), [34566](#)
- _text_change_case_breathing_-
aux:nnnnnn [34507](#), [34540](#), [34549](#)
- _text_change_case_breathing_-
aux:nnnnw [34507](#), [34554](#), [34557](#)
- _text_change_case_breathing_-
dialytika:nnnn [34507](#), [34571](#), [34573](#)
- _text_change_case_catcode:nn ..
..... [33377](#), [33855](#), [33870](#),
[33874](#), [33952](#), [34119](#), [34123](#), [34502](#),
[34589](#), [34591](#), [34602](#), [34604](#), [34664](#),
[34666](#), [34668](#), [34678](#), [34712](#), [34735](#),
[34811](#), [34823](#), [34844](#), [34849](#), [34858](#)

- _text_change_case_codepoint:nn
..... 33377, 33820,
33823, 34006, 34015, 34097, 34109,
34132, 34141, 34153, 34177, 34560
- _text_change_case_codepoint:nnn
..... 33377,
33825, 33828, 33833, 33837, 33838
- _text_change_case_codepoint:nnnn
..... 33377, 33750, 33757, 33810,
33814, 33956, 33991, 34583, 34595,
34608, 34611, 34622, 34632, 34675,
34732, 34767, 34779, 34814, 34861
- _text_change_case_codepoint_
aux:nn 33377, 33830, 33845
- _text_change_case_codepoint_
aux:nnn 33377, 33836, 33842
- _text_change_case_codepoint_
aux:nnnn 33847, 33849
- _text_change_case_codepoint_
lower:nnnn 33377, 33741
- _text_change_case_codepoint_
title:nnn 33377, 33793, 33799, 33803
- _text_change_case_codepoint_
title:nnnn 33377, 33783
- _text_change_case_codepoint_
title_auxi:nnnn 33377, 33787, 33796
- _text_change_case_codepoint_
title_auxii:nnnn
..... 33377, 33800, 33804, 33805
- _text_change_case_codepoint_
upper:nnnn 33377, 33747
- _text_change_case_cs_check:nnnN
..... 33377, 33518, 33563
- _text_change_case_custom:nnnnn
..... 33377
- _text_change_case_custom:nnnnnn
..... 33712, 33719, 33721, 33725
- _text_change_case_custom_
lower:nnnn ... 33377, 33710, 33716
- _text_change_case_custom_
title:nnnn 33377, 33717
- _text_change_case_custom_
upper:nnnn 33377, 33715
- _text_change_case_exclude:nnnN
..... 33377, 33566, 33573
- _text_change_case_exclude:nnnn
..... 33377, 33576, 33588
- _text_change_case_exclude:nnnNN
..... 33377, 33595, 33598, 33607
- _text_change_case_exclude:nnnnN
..... 33377, 33581, 33593
- _text_change_case_exclude:nnnNnn
..... 33377, 33610, 33611
- _text_change_case_exclude:nnnNw
..... 33377, 33605, 33609
- _text_change_case_exclude_
aux:nnnN 33377, 33577, 33579
- _text_change_case_exclude_
word:n 33377,
33433, 33436, 33490, 33492, 33575
- _text_change_case_exclude_
words:nn 33377, 33425, 33430
- _text_change_case_generate:n ..
..... 33945, 33945, 34161, 34183
- _text_change_case_group_
lower:nnnn ... 33377, 33457, 33464
- _text_change_case_group_
title:nnnn 33377, 33465
- _text_change_case_group_
upper:nnnn 33377, 33463
- _text_change_case_if_greek:n ..
..... 33960, 34241, 34243, 34246
- _text_change_case_if_greek:nTF
..... 33960, 34509
- _text_change_case_if_greek_
accent:n 33960, 34270, 34272, 34275
- _text_change_case_if_greek_
accent:nTF 33960, 34079
- _text_change_case_if_greek_
accent_p:n 34060, 34236
- _text_change_case_if_greek_
breathing:n
..... 33960, 34381, 34384, 34387
- _text_change_case_if_greek_
breathing:nTF 33960, 34082
- _text_change_case_if_greek_
breathing_p:n 34061, 34237
- _text_change_case_if_greek_p:n
..... 33963
- _text_change_case_if_greek_
spacing_diacritic:n
..... 33960, 34303, 34306, 34309
- _text_change_case_if_greek_
spacing_diacritic:nTF 33960, 33970
- _text_change_case_if_greek_
stress:n 33960, 34399, 34402, 34405
- _text_change_case_if_greek_
stress:nTF 33960, 34091
- _text_change_case_if_takes_
dialytika:n
..... 33960, 34417, 34419, 34422
- _text_change_case_if_takes_
dialytika:nTF
..... 33960, 34107, 34151, 34575
- _text_change_case_if_takes_
ypogegrammeni:n
..... 33960, 34442, 34444, 34447

`\_text_change_case_if_takes_-`
`ypogegrammeni:nTF` ... [33960](#), [34019](#)
`\_text_change_case_inner:nnnn` ...
[33377](#),
[33422](#), [33437](#), [33460](#), [33468](#), [33619](#)
`\_text_change_case_letterlike:nnnnnN`
... [33377](#), [33685](#), [33689](#), [33690](#)
`\_text_change_case_letterlike_-`
`lower:nnnN` ... [33377](#), [33684](#), [33687](#)
`\_text_change_case_letterlike_-`
`title:nnnN` ... [33377](#), [33688](#)
`\_text_change_case_letterlike_-`
`upper:nnnN` ... [33377](#), [33686](#)
`\_text_change_case_loop:nnnw` ...
... [33377](#), [33440](#), [33444](#), [33461](#),
[33480](#), [33546](#), [33591](#), [33623](#), [33635](#),
[33647](#), [33652](#), [33707](#), [33766](#), [33781](#),
[33886](#), [33973](#), [33989](#), [34007](#), [34016](#),
[34088](#), [34094](#), [34098](#), [34133](#), [34142](#),
[34220](#), [34226](#), [34239](#), [34468](#), [34476](#),
[34499](#), [34503](#), [34518](#), [34529](#), [34555](#),
[34563](#), [34578](#), [34580](#), [34669](#), [34686](#),
[34714](#), [34812](#), [34824](#), [34845](#), [34850](#)
`\_text_change_case_lower_-`
`az:nnnnn` ... [34863](#), [34863](#)
`\_text_change_case_lower_-`
`la-x-medieval:nnnnn` ... [34614](#)
`\_text_change_case_lower_-`
`lt:nnnN` ... [34634](#), [34685](#), [34689](#)
`\_text_change_case_lower_-`
`lt:nnnn` ... [34634](#), [34692](#), [34694](#)
`\_text_change_case_lower_-`
`lt:nnnnn` ... [34634](#)
`\_text_change_case_lower_-`
`lt:nnnw` ... [34634](#), [34679](#), [34682](#)
`\_text_change_case_lower_lt_-`
`auxi:nnnnn` ... [34636](#), [34647](#)
`\_text_change_case_lower_lt_-`
`auxii:nnnnn` ... [34651](#), [34672](#)
`\_text_change_case_lower_-`
`sigma:nnnnN` ... [33377](#), [33763](#), [33770](#)
`\_text_change_case_lower_-`
`sigma:nnnnn` ... [33377](#), [33744](#), [33753](#)
`\_text_change_case_lower_-`
`sigma:nnnnw` ... [33377](#), [33756](#), [33760](#)
`\_text_change_case_lower_-`
`tr:NnnnN` ... [34803](#), [34820](#), [34828](#)
`\_text_change_case_lower_-`
`tr:Nnnnn` ... [34803](#), [34831](#), [34833](#)
`\_text_change_case_lower_-`
`tr:nnnnn` ... [34803](#), [34803](#), [34864](#)
`\_text_change_case_lower_-`
`tr:nnnNw` ... [34803](#), [34806](#), [34817](#)
`\_text_change_case_math_-`
`group:nnnNn` ... [33377](#), [33535](#), [33549](#)
`\_text_change_case_math_-`
`loop:nnnNw` ... [33377](#),
[33524](#), [33529](#), [33547](#), [33552](#), [33561](#)
`\_text_change_case_math_N_-`
`type:nnnNN` ... [33377](#), [33532](#), [33540](#)
`\_text_change_case_math_-`
`search:nnnNNN` ...
... [33377](#), [33511](#), [33515](#), [33527](#)
`\_text_change_case_math_-`
`space:nnnNw` ... [33377](#), [33536](#), [33556](#)
`\_text_change_case_N_type:nnnN` ...
... [33377](#), [33447](#), [33498](#)
`\_text_change_case_N_type:nnnnN`
... [33377](#), [33506](#), [33509](#)
`\_text_change_case_N_type_-`
`aux:nnnN` ... [33377](#), [33502](#), [33504](#)
`\_text_change_case_next_end:nnn`
... [33377](#), [33891](#)
`\_text_change_case_next_-`
`lower:nnn` [33377](#), [33885](#), [33888](#), [33890](#)
`\_text_change_case_next_-`
`title:nnn` ... [33377](#), [33889](#)
`\_text_change_case_next_-`
`upper:nnn` ... [33377](#), [33887](#)
`\_text_change_case_replace:nnnN`
... [33377](#), [33601](#), [33625](#)
`\_text_change_case_replace:nnnn`
... [33377](#),
[33629](#), [33634](#), [33636](#), [33729](#), [33735](#)
`\_text_change_case_setup:NN` ...
... [34868](#), [34875](#), [34877](#)
`\_text_change_case_setup:Nn` ...
... [34901](#), [34921](#), [34923](#)
`\_text_change_case_skip:nnw` ...
... [33377](#), [33469](#),
[33657](#), [33659](#), [33675](#), [33680](#), [33892](#)
`\_text_change_case_skip_-`
`group:nnn` ... [33377](#), [33665](#), [33677](#)
`\_text_change_case_skip_N_-`
`type:nnN` ... [33377](#), [33662](#), [33670](#)
`\_text_change_case_skip_-`
`space:nnw` ... [33377](#), [33666](#), [33682](#)
`\_text_change_case_space:nnnw` ...
... [33377](#), [33451](#), [33473](#), [33683](#)
`\_text_change_case_space_-`
`break:nnn` ... [33377](#), [33484](#)
`\_text_change_case_space_-`
`break:w` ... [33377](#), [33485](#), [33486](#)
`\_text_change_case_space_break_-`
`aux:w` ... [33377](#), [33489](#), [33492](#), [33496](#)
`\_text_change_case_switch:nnnN` ...
... [33377](#), [33632](#), [33637](#)

- _text_change_case_switch_-
lower:nnnNnnnn [33377](#), [33644](#)
- _text_change_case_switch_-
title:nnnNnnnn [33377](#), [33654](#)
- _text_change_case_switch_-
upper:nnnNnnnn [33377](#), [33649](#)
- _text_change_case_title_-
el:nnnnn [34582](#), [34582](#)
- _text_change_case_title_-
hy-x-yiwn:nnnnn [34584](#)
- _text_change_case_title_-
hy:nnnnn [34584](#), [34597](#)
- _text_change_case_title_-
nl:nnnN [34763](#), [34784](#), [34788](#)
- _text_change_case_title_-
nl:nnnnn [34763](#), [34763](#)
- _text_change_case_title_-
nl:nnnw [34763](#), [34777](#), [34781](#)
- _text_change_case_title_nl_-
aux:nnnnn [34763](#), [34766](#), [34770](#)
- _text_change_case_upper_-
az:nnnnn [34863](#), [34865](#)
- _text_change_case_upper_-
de-alt:nnnnn [33947](#)
- _text_change_case_upper_-
de-x-eszett:nnnnn [33947](#)
- _text_change_case_upper_-
el-x-iota:nnnnn [33960](#)
- _text_change_case_upper_-
el-x-iota_ypogegrammeni:n . [33960](#)
- _text_change_case_upper_-
el:nnnn
.. [33960](#), [33976](#), [33996](#), [34083](#), [34157](#)
- _text_change_case_upper_-
el:nnnnN [33960](#), [34004](#), [34011](#)
- _text_change_case_upper_-
el:nnnnn [33960](#), [33960](#), [33995](#)
- _text_change_case_upper_-
el:nnnnw [33960](#), [33999](#), [34001](#)
- _text_change_case_upper_el_-
aux:nnnnN [33960](#),
[34024](#), [34035](#), [34041](#), [34066](#), [34069](#)
- _text_change_case_upper_el_-
aux:nnnnn [33960](#), [34072](#), [34074](#)
- _text_change_case_upper_el_-
dialytika:n
.. [33960](#), [34108](#), [34112](#), [34154](#), [34577](#)
- _text_change_case_upper_el_-
dialytika:nnnn [33960](#), [34077](#), [34105](#)
- _text_change_case_upper_el_-
gobble:nnnN ... [33960](#), [34219](#), [34223](#)
- _text_change_case_upper_el_-
gobble:nnnn ... [33960](#), [34229](#), [34233](#)
- _text_change_case_upper_el_-
gobble:nnnw
.. [33960](#), [34110](#), [34155](#), [34215](#), [34238](#)
- _text_change_case_upper_el_-
hiatus:nnnnN .. [33960](#), [34130](#), [34137](#)
- _text_change_case_upper_el_-
hiatus:nnnnn .. [33960](#), [34146](#), [34149](#)
- _text_change_case_upper_el_-
hiatus:nnnnw .. [33960](#), [34080](#), [34126](#)
- _text_change_case_upper_el_-
stress:nn [33960](#), [34093](#), [34181](#)
- _text_change_case_upper_el_-
ypogegrammeni:n [33960](#), [34159](#)
- _text_change_case_upper_el_-
ypogegrammeni:nnnnnnN
..... [33960](#), [34032](#), [34038](#)
- _text_change_case_upper_el_-
ypogegrammeni:nnnnnnn
..... [33960](#), [34045](#), [34051](#)
- _text_change_case_upper_el_-
ypogegrammeni:nnnnnnw
.. [33960](#), [34021](#), [34027](#), [34055](#), [34063](#)
- _text_change_case_upper_-
hy-x-yiwn:nnnnn [34584](#)
- _text_change_case_upper_-
hy:nnnnn [34584](#), [34584](#)
- _text_change_case_upper_-
la-x-medieval:nnnnn [34614](#)
- _text_change_case_upper_-
lt:nnnN [34716](#), [34742](#), [34746](#)
- _text_change_case_upper_-
lt:nnnn [34716](#), [34749](#), [34751](#)
- _text_change_case_upper_-
lt:nnnnn [34716](#)
- _text_change_case_upper_-
lt:nnnw [34716](#), [34736](#), [34739](#)
- _text_change_case_upper_lt_-
aux:nnnnn [34718](#), [34729](#)
- _text_change_case_upper_-
tr:nnnnn [34853](#), [34853](#), [34866](#)
- _text_change_cases_lower_-
lt:nnnnn [34634](#)
- _text_change_cases_lower_lt_-
auxi:nnnnn [34634](#)
- _text_change_cases_lower_lt_-
auxii:nnnnn [34634](#)
- _text_change_cases_upper_-
lt:nnnnn [34716](#)
- _text_change_cases_upper_lt_-
aux:nnnnn [34716](#)
- _text_char_catcode:N ... [32776](#),
[32776](#), [33765](#), [33779](#), [33780](#), [33871](#),
[33877](#), [34618](#), [34628](#), [34776](#), [34798](#)

- \c__text_chardef_group_begin_-token [32981](#)
- \c__text_chardef_group_end_token [32981](#)
- \c__text_chardef_space_token . [32981](#)
- __text_codepoint_compare:nNn [32864](#), [32873](#)
- __text_codepoint_compare:nNnTF [32861](#), [33755](#), [33851](#),
[33876](#), [33949](#), [33985](#), [34053](#), [34076](#),
[34085](#), [34586](#), [34599](#), [34616](#), [34626](#),
[34805](#), [34808](#), [34855](#), [35058](#), [35064](#)
- __text_codepoint_compare_p:nNn [32861](#), [33966](#), [33967](#), [34115](#), [34116](#),
[34487](#), [34488](#), [34489](#), [34490](#), [34552](#),
[34553](#), [34705](#), [34706](#), [34707](#), [34759](#),
[34841](#), [35173](#), [35174](#), [35212](#), [35213](#)
- __text_codepoint_from_chars:N [32861](#), [32891](#), [32899](#), [32918](#)
- __text_codepoint_from_chars:NN [32861](#), [32906](#), [32919](#)
- __text_codepoint_from_chars:NNN [32861](#), [32910](#), [32921](#)
- __text_codepoint_from_chars:NNNN [32861](#), [32913](#), [32923](#)
- __text_codepoint_from_chars:Nw [32861](#), [32869](#), [32875](#), [32879](#),
[33790](#), [33826](#), [33979](#), [34164](#), [34186](#),
[34190](#), [34202](#), [34244](#), [34273](#), [34307](#),
[34385](#), [34403](#), [34420](#), [34445](#), [34514](#),
[34638](#), [34653](#), [34720](#), [35100](#), [35181](#),
[35226](#), [35284](#), [35341](#), [35385](#), [35451](#)
- __text_codepoint_from_chars_-aux:Nw . . . [32861](#), [32885](#), [32895](#), [32903](#)
- __text_codepoint_process:nN [32823](#), [32825](#), [32828](#), [33568](#), [33998](#),
[34043](#), [34071](#), [34145](#), [34228](#), [34478](#),
[34522](#), [34531](#), [34544](#), [34570](#), [34691](#),
[34748](#), [34830](#), [35040](#), [35128](#), [35274](#)
- __text_codepoint_process:nNN [32823](#), [32846](#), [32854](#)
- __text_codepoint_process:nNNN [32823](#), [32849](#), [32856](#)
- __text_codepoint_process:nNNNN [32823](#), [32850](#), [32858](#)
- __text_codepoint_process_aux:nN [32823](#), [32833](#), [32837](#), [32843](#)
- __text_declare_case_exclusion:n [33931](#), [33932](#), [33934](#), [33936](#), [33937](#)
- __text_declare_case_mapping:n [33898](#), [33899](#), [33901](#), [33903](#), [33904](#)
- __text_declare_case_mapping:nnnn [33898](#), [33915](#), [33917](#), [33919](#), [33920](#)
- __text_declare_case_mapping_-aux:nnn [33898](#), [33906](#), [33909](#)
- __text_declare_case_mapping_-aux:nnnn [33898](#), [33922](#), [33925](#)
- __text_end_env:n . [35731](#), [35732](#), [35733](#)
- __text_expand:n [32987](#), [32992](#), [32995](#), [33031](#)
- __text_expand_accent:N [32987](#), [33150](#), [33165](#)
- __text_expand_accent:NN [32987](#), [33167](#), [33171](#), [33183](#)
- __text_expand_cs:N [32987](#), [33194](#), [33205](#)
- __text_expand_cs_expand:N [32987](#), [33283](#), [33286](#)
- __text_expand_encoding:N [32987](#), [33254](#), [33261](#)
- __text_expand_encoding_escape:N [32987](#)
- __text_expand_encoding_escape:NN [33266](#), [33269](#)
- __text_expand_end:w [32987](#), [33007](#), [33044](#), [33078](#), [33232](#)
- __text_expand_exclude:N [32987](#), [33099](#), [33123](#)
- __text_expand_exclude:NN [32987](#), [33144](#), [33147](#), [33156](#)
- __text_expand_exclude:nN [32987](#), [33128](#), [33142](#)
- __text_expand_exclude:Nnn [32987](#), [33159](#), [33160](#)
- __text_expand_exclude:Nw [32987](#), [33154](#), [33158](#)
- __text_expand_exclude_switch:Nnnnn [32987](#), [33126](#), [33137](#)
- __text_expand_explicit:N [32987](#), [33053](#), [33096](#)
- __text_expand_group:n [32987](#), [33019](#), [33024](#)
- __text_expand_letterlike:N [32987](#), [33174](#), [33185](#)
- __text_expand_letterlike:NN [32987](#), [33187](#), [33191](#), [33203](#)
- __text_expand_loop:w [32987](#), [32998](#), [33013](#), [33034](#), [33039](#),
[33082](#), [33114](#), [33119](#), [33140](#), [33163](#),
[33180](#), [33200](#), [33223](#), [33248](#), [33259](#),
[33266](#), [33285](#), [33292](#), [33296](#), [33324](#)
- __text_expand_math_group:Nn [32987](#), [33070](#), [33085](#)
- __text_expand_math_loop:Nw [32987](#), [33059](#), [33064](#), [33083](#), [33088](#), [33094](#)
- __text_expand_math_N_type:NN [32987](#), [33067](#), [33075](#)

`\__text_expand_math_search:NNN` ..
 32987, 33045, 33050, 33062
`\__text_expand_math_space:Nw` ...
 32987, 33071, 33090
`\__text_expand_N_type:N`
 32987, 33016, 33041
`\__text_expand_protect:N`
 32987, 33220, 33227
`\__text_expand_protect:nN`
 32987, 33234, 33237
`\__text_expand_protect:Nw`
 32987, 33238, 33239
`\__text_expand_protect:w`
 32987, 33208, 33217
`\__text_expand_replace:N`
 32987, 33214, 33267, 33270
`\__text_expand_replace:n`
 32987, 33280, 33285
`\__text_expand_result:n`
 .. 32987, 33000, 33005, 33006, 33007
`\__text_expand_space:w`
 32987, 33020, 33036
`\__text_expand_store:n`
 32987, 33002, 33004,
 33026, 33038, 33058, 33080, 33087,
 33093, 33117, 33118, 33139, 33162,
 33179, 33199, 33222, 33231, 33244,
 33245, 33247, 33258, 33295, 33323
`\__text_expand_store:nw`
 32987, 33003, 33005
`\__text_expand_testopt:N`
 32987, 33213, 33250
`\__text_expand_testopt:NNn`
 32987, 33253, 33256
`\__text_expand_unexpanded:N`
 1352, 32987, 33310, 33314
`\__text_expand_unexpanded:n`
 32987, 33307, 33321
`\__text_expand_unexpanded:w`
 32987, 33291, 33299, 33309
`\__text_expand_unexpanded_test:w`
 32987, 33301, 33304
`\__text_if_expandable:N` 32808
`\__text_if_expandable:NTF`
 32808, 33288, 35702
`\__text_if_q_recursion_tail_-`
 stop_do:Nn 32670, 32670, 33052,
 33149, 33173, 33193, 33500, 33517,
 33542, 33600, 33672, 34568, 34970,
 34986, 35011, 35081, 35118, 35263,
 35552, 35564, 35605, 35633, 35687
`\__text_if_q_recursion_tail_-`
 stop_do:nn ... 32670, 32671, 33495
`\__text_if_recursion_tail_stop:N`
 35505, 35505
`\__text_if_s_recursion_tail_-`
 stop_do:Nn
 .. 32676, 32676, 33043, 33077, 33229
`\__text_loop:Nn` 35749,
 35757, 35759, 35802, 35807, 35809
`\__text_loop:NNn` . 35828, 35834, 35836
`\__text_map_active_check:nnnn` ...
 34937, 35041, 35044
`\__text_map_ALetter:nnnn` 35321, 35406
`\__text_map_class:nnnn`
 34937, 35066, 35095
`\__text_map_class:nnnnn`
 34937, 35097, 35103
`\__text_map_codepoint:nnnn`
 34937, 35054, 35056
`\__text_map_collect:nnnnn`
 35321, 35331, 35363,
 35409, 35417, 35425, 35430, 35438
`\__text_map_collect_auxi:nnnnnnnn`
 35321, 35334, 35336
`\__text_map_collect_auxii:nnnnnnnn`
 35321, 35338, 35345
`\__text_map_collect_auxiii:n` ...
 .. 35321, 35354, 35366, 35378, 35392
`\__text_map_collect_auxiv:nnnnnnnn`
 35321, 35369, 35380, 35398
`\__text_map_collect_auxv:nnnnnnnn`
 35321, 35382, 35389
`\__text_map_Control:nnnn`
 35148, 35148, 35154
`\__text_map_CR:nnnN`
 34937, 35072, 35079
`\__text_map_CR:nnnw`
 34937, 35061, 35069
`\__text_map_cs_check:nnnN`
 34937, 34987, 35031
`\__text_map_Extend:nnnn`
 35148, 35155, 35157, 35158
`\__text_map_ExtendNumLet:nnnn` ...
 35321, 35440
`\__text_map_ExtendNumLet_-`
 auxi:nnnnn 35321
`\__text_map_ExtendNumLet_-`
 auxi:nnnnn ... 35444, 35446, 35474
`\__text_map_ExtendNumLet_-`
 auxii:nnnn ... 35321, 35448, 35455
`\__text_map_Format:nnnn` 35148, 35157
`\__text_map_group:nnnn`
 34937, 34950, 34955
`\__text_map_hangul:nnnN`
 35193, 35254, 35261

- _text_map_hangul:nnnn [35193](#), [35275](#), [35279](#)
- _text_map_hangul:nnnnnw [35292](#), [35295](#)
- _text_map_hangul:nnnnw [35193](#)
- _text_map_hangul:nnnw [35193](#), [35234](#), [35240](#), [35247](#), [35251](#), [35306](#), [35311](#), [35317](#)
- _text_map_hangul_aux:nnnnw ... [35193](#), [35281](#), [35288](#), [35294](#), [35302](#)
- _text_map_hangul_end:nw [35193](#), [35303](#)
- _text_map_hangul_L:nnn [35193](#), [35304](#)
- _text_map_hangul_LV:nnn [35193](#), [35309](#), [35314](#)
- _text_map_hangul_LVT:nnn [35193](#), [35315](#), [35320](#)
- _text_map_hangul_next:nnnnn ... [35193](#), [35299](#), [35301](#)
- _text_map_hangul_T:nnn [35193](#), [35320](#)
- _text_map_hangul_V:nnn [35193](#), [35314](#)
- _text_map_Hebrew_Letter:nnnn ... [35321](#), [35414](#)
- _text_map_if_ignorable:n ... [35131](#)
- _text_map_if_ignorable:nTF ... [34937](#), [35178](#), [35362](#), [35395](#), [35471](#)
- _text_map_Katakana:nnnn [35321](#), [35422](#)
- _text_map_L:nnnn [35193](#), [35231](#)
- _text_map_lookahead:nnnnN ... [34937](#), [35112](#), [35116](#)
- _text_map_lookahead:nnnnnw ... [34937](#), [35109](#), [35167](#), [35184](#), [35206](#), [35333](#), [35368](#), [35397](#), [35443](#), [35473](#)
- _text_map_loop:nnnw [34937](#), [34939](#), [34944](#), [34959](#), [34966](#), [35019](#), [35037](#), [35052](#), [35065](#), [35075](#), [35091](#), [35093](#), [35113](#), [35126](#), [35152](#), [35156](#), [35162](#), [35187](#), [35189](#), [35191](#), [35216](#), [35228](#), [35229](#), [35257](#), [35271](#), [35291](#), [35373](#), [35402](#), [35478](#)
- _text_map_LV:nnnn [35193](#), [35237](#), [35243](#)
- _text_map_LVT:nnnn [35193](#), [35244](#), [35250](#)
- _text_map_math_group:nnnNn ... [34937](#), [35004](#), [35023](#)
- _text_map_math_loop:nnnNw [34937](#), [34993](#), [34998](#), [35021](#), [35024](#), [35030](#)
- _text_map_math_N_type:nnnNN ... [34937](#), [35001](#), [35009](#)
- _text_map_math_search:nnnnN ... [34937](#), [34975](#), [34978](#)
- _text_map_math_search:nnnNNN ... [34937](#), [34980](#), [34984](#), [34996](#)
- _text_map_math_space:nnnNw ... [34937](#), [35005](#), [35027](#)
- _text_map_N_type:nnnN [34937](#), [34947](#), [34968](#)
- _text_map_Newline:nnnn [35148](#), [35154](#)
- _text_map_Numeric:nnnn [35321](#), [35427](#)
- _text_map_Other:nnnn [35106](#), [35148](#), [35159](#), [35358](#), [35467](#)
- _text_map_output:nn [34937](#), [34957](#), [34958](#), [34964](#), [34972](#), [34992](#), [35013](#), [35018](#), [35035](#), [35050](#), [35060](#), [35090](#), [35142](#), [35150](#), [35151](#), [35161](#), [35166](#), [35205](#), [35233](#), [35239](#), [35246](#), [35372](#), [35401](#), [35408](#), [35416](#), [35424](#), [35429](#), [35437](#), [35442](#), [35477](#)
- _text_map_Prepend:nnn [35193](#), [35218](#), [35220](#)
- _text_map_Prepend:nnnn [35193](#), [35203](#)
- _text_map_Prepend_aux:nnnnn ... [35193](#), [35207](#), [35209](#)
- _text_map_Regional_Indicator:nnnn [35148](#), [35164](#)
- _text_map_Regional_Indicator_aux:nnnnn [35148](#), [35168](#), [35170](#), [35185](#)
- _text_map_space:nnnw [34937](#), [34951](#), [34962](#)
- _text_map_SpacingMark:nnnn ... [35148](#), [35158](#)
- _text_map_T:nnnn [35193](#), [35250](#)
- _text_map_tokens:nnn [33424](#), [34937](#), [34937](#), [34943](#), [35195](#), [35200](#), [35323](#), [35328](#)
- _text_map_V:nnnn [35193](#), [35243](#)
- _text_map_WSegSpace:nnnn [35321](#), [35435](#)
- \l_text_math_mode_tl [32980](#)
- \c_text_mathchardef_group_begin_token [32981](#)
- \c_text_mathchardef_group_end_token [32981](#)
- \c_text_mathchardef_space_token [32981](#)
- _text_purify:n . [35506](#), [35511](#), [35516](#)
- _text_purify_accent:NN [35786](#), [35786](#), [35800](#)
- _text_purify_encoding:N [35506](#), [35683](#), [35695](#)
- _text_purify_encoding_escape:NN [35506](#), [35700](#), [35707](#)
- _text_purify_end:w [35506](#), [35527](#), [35552](#), [35590](#), [35687](#)

- _text_purify_expand:N 35506, 35673, 35679
- _text_purify_group:n 35506, 35539, 35544
- _text_purify_loop:w 35506, 35519, 35533, 35544, 35548, 35585, 35661, 35668, 35676, 35693, 35703, 35704, 35710
- _text_purify_math_cmd:N 35506, 35565, 35626
- _text_purify_math_cmd:n 35638, 35642
- _text_purify_math_cmd:NN 35506, 35628, 35631, 35640
- _text_purify_math_cmd:Nn ... 35506
- _text_purify_math_end:w 35506, 35582, 35608, 35643
- _text_purify_math_group:NNn ... 35506, 35598, 35614
- _text_purify_math_loop:NNw ... 35506, 35575, 35592, 35611, 35617, 35624
- _text_purify_math_N_type:NNN .. 35506, 35595, 35603
- _text_purify_math_result:n ... 35576, 35580, 35581, 35582, 35587, 35643
- _text_purify_math_search:NNN .. 35506, 35557, 35562, 35571
- _text_purify_math_space:NNw ... 35506, 35599, 35619
- _text_purify_math_start:NNw ... 35506, 35569, 35573
- _text_purify_math_stop:Nw 35587, 35606
- _text_purify_math_store:n 35506, 35578, 35610, 35616, 35623
- _text_purify_math_store:nw ... 35506, 35579, 35580
- _text_purify_N_type:N 35506, 35536, 35550
- _text_purify_N_type_aux:N 35506, 35553, 35555
- _text_purify_protect:N 35506, 35682, 35685
- _text_purify_replace:N 35506, 35634, 35644
- _text_purify_replace_auxi:n ... 35506, 35654, 35668, 35669, 35690
- _text_purify_replace_auxii:n .. 35506, 35663, 35670
- _text_purify_result:n 35521, 35525, 35526, 35527
- _text_purify_space:w 35506, 35540, 35545
- _text_purify_store:n 35506, 35523, 35547, 35584, 35589, 35675, 35709
- _text_purify_store:nw 35506, 35524, 35525
- _text_quark_if_nil:n 32665
- _text_quark_if_nil:nTF 32665, 33241
- _text_quark_if_nil_p:n 32665
- _text_sep: 32684, 32684, 32757, 32760
- _text_tmp:w 32956, 32974
- _text_token_to_explicit:N 32685, 32687, 35664
- _text_token_to_explicit:n 32685, 32739, 32743
- _text_token_to_explicit_auxi:w 32685, 32745, 32760
- _text_token_to_explicit_-auxii:w 32685, 32765, 32773
- _text_token_to_explicit_-auxiii:w 32685, 32767, 32775
- _text_token_to_explicit_char:N 32685, 32697, 32729
- _text_token_to_explicit_cs:N .. 32685, 32695, 32702
- _text_token_to_explicit_cs_-aux:N 32685, 32706, 32712
- _text_use_i_delimit_by_q_-recursion_stop:nw 32668, 32668, 33056, 33153, 33177, 33197, 33521, 33604, 34990, 35568, 35637
- _text_use_i_delimit_by_s_-recursion_stop:nw 32674, 32674, 32681
- \textbaselineshiftfactor 1189
- \textdir 922
- \textdirection 923
- \textfont 418
- \textstyle 419
- \texttt 40527
- \TeXeTstate 528
- \tfont 1191
- \TH 33347, 34888, 35769
- \th 33347, 34888, 35782
- \the 29, 85, 86, 87, 88, 89, 90, 91, 92, 93, 420
- \thickmuskip 421
- \thinmuskip 422
- \time 423, 1281, 9095, 9097
- \tiny 38409
- tl commands:
 - \c_catcode_active_space_tl 202, 20084
 - \c_catcode_other_space_tl 203, 675,

- 10711, 10755, 10836, 10925, 11001,
16663, 16675, 16857, 16913, 20089
- \c_empty_tl 130, 896,
910, 9584, 12249, 12260, 12262,
12297, 12778, 13618, 13660, 13673,
14430, 18797, 18803, 19180, 19196
- \c_novalue_tl . . 116, 130, 12298, 12883
- \c_space_tl 131, 3616, 3646,
3964, 9175, 9384, 9672, 9674, 11722,
12302, 13340, 14310, 19686, 23791,
32455, 32565, 32968, 33036, 33091,
33473, 33476, 33557, 33560, 34962,
35028, 35545, 35621, 35744, 38188,
38232, 38299, 38726, 38727, 38728,
38735, 38736, 38911, 38912, 38918,
38919, 38920, 39764, 39877, 39920,
39921, 39943, 41329, 41330, 41338,
41339, 41361, 41362, 41367, 41368
- \tl_analysis_log:N . . . 48, 3994, 3996
- \tl_analysis_log:n . . . 48, 4011, 4013
- \tl_analysis_map_inline:Nn
. 48, 3970, 3970, 5886
- \tl_analysis_map_inline:nn
. 48, 214, 555,
589, 590, 3970, 3971, 3972, 6662, 7485
- \tl_analysis_show:N . . 48, 3994, 3994
- \tl_analysis_show:n . . 48, 4011, 4011
- \tl_build_begin:N 132,
133, 536, 754, 4979, 5488, 6063,
6164, 6694, 6728, 6900, 6966, 7903,
13584, 13584, 41592, 41593, 42206
- \tl_build_clear:N 41592, 41593
- \tl_build_end:N 132, 133,
536, 754, 5009, 5017, 5498, 6120,
6183, 7000, 7929, 7992, 13648, 13648
- \tl_build_gbegin:N 132,
133, 13584, 13586, 41594, 41595, 42288
- \tl_build_gclear:N 41592, 41595
- \tl_build_gend:N 132, 133, 13648, 13653
- \tl_build_get:NN 41596, 41597
- \tl_build_get_intermediate:NN
133, 6719, 13666, 13666, 41596, 41597
- \tl_build_gput_left:Nn
. 132, 13631, 13634, 13636, 42290
- \tl_build_gput_right:Nn
. 132, 13599, 13605, 13610, 42289
- \tl_build_put_left:Nn
. 132, 13631, 13631, 13633, 42208
- \tl_build_put_right:Nn
. 132, 567, 755, 4986,
5004, 5012, 5016, 5080, 5083, 5123,
5137, 5141, 5264, 5278, 5319, 5341,
5354, 5386, 5399, 5403, 5485, 5491,
5497, 5501, 5544, 5834, 5838, 5845,
5851, 5872, 5888, 5906, 6134, 6179,
6192, 6762, 6989, 7054, 7123, 7180,
7183, 7197, 7265, 7907, 8008, 8011,
8019, 8022, 13599, 13599, 13604, 42207
- \tl_case:Nn 41581, 41582, 41589, 41590
- \tl_case:NnTF
. 41581, 41584, 41586, 41588
- \tl_clear:N . . 114, 4307, 4652, 6695,
6729, 9165, 10804, 10805, 10808,
10818, 10928, 10931, 10991, 12259,
12259, 12263, 12266, 12477, 13651,
14711, 19239, 19240, 22905, 23253,
23378, 32958, 39381, 39810, 42194
- \tl_clear_new:N 114,
11662, 11663, 11664, 11665, 11666,
12265, 12265, 12269, 19243, 19244,
33328, 33895, 33911, 33927, 33929,
33938, 35714, 39390, 39424, 39465
- \tl_concat:NNN
114, 12277, 12277, 12293, 13714, 42131
- \tl_const:Nn 114, 603, 3610,
4345, 4432, 7334, 8956, 9312, 9313,
9348, 9353, 9355, 9357, 9359, 9361,
9366, 9367, 9374, 10701, 10707,
11129, 12252, 12252, 12257, 12258,
12297, 12300, 12302, 12468, 14546,
14551, 17382, 17437, 19236, 20062,
20087, 20089, 20161, 20803, 24244,
24245, 24246, 24247, 24248, 24256,
24345, 26462, 27861, 28318, 28320,
28322, 28324, 28326, 28328, 28330,
28332, 28334, 32244, 32284, 32498,
32523, 32551, 34871, 34873, 34891,
34892, 34909, 34916, 35805, 35831,
39534, 39535, 39536, 39537, 39538,
39583, 39584, 39585, 39595, 39596,
39597, 39598, 39599, 39600, 39601,
39602, 39721, 39736, 39750, 39784,
40040, 40045, 40050, 40211, 42313
- \tl_count:N
. . . 35, 116, 119, 13016, 13021, 13028
- \tl_count:n . . 35, 116, 119, 414, 772,
882, 1070, 1420, 1641, 1645, 2107,
2158, 5750, 7415, 7419, 7438, 7442,
7512, 7516, 13016, 13016, 13027,
13393, 13408, 13422, 36168, 36297
- \tl_count_tokens:n
. 119, 13029, 13029, 13042
- \tl_format:Nn 127, 16759, 16759, 16760
- \tl_format:nn 127, 16759, 16759, 16761
- \tl_gclear:N . . 114, 451, 3298, 6968,
9064, 9240, 12259, 12261, 12264,
12268, 13656, 19241, 19242, 42276

- \tl_gclear_new:N
 114, [12265](#), 12267, 12270, 19245, 19246
- \tl_gconcat:NNN
 114, [12277](#), 12285, 12294, 13715, 42132
- \tl_gput_left:Nn
 . 114, [12319](#), 12350, 12355, 12360,
 12365, [12373](#), 12387, 12388, 12389,
 12390, 12391, 12392, 15437, 15620,
 42278, 42279, 42280, 42281, 42282
- \tl_gput_right:Nn
 115, 1585, 1586, 6998,
 7049, 7050, 7126, 8947, 8949, 9067,
 9243, [12393](#), 12421, 12423, 12428,
 12433, 12441, 12455, 12456, 12457,
 12458, 12459, 12460, 17352, 17569,
 31132, 31820, 32951, 32953, 34896,
 34898, 40818, 40878, 40880, 41135,
 42283, 42284, 42285, 42286, 42287
- \tl_gremove_all:Nn
 129, [12768](#), 12770, 12774, 12775
- \tl_gremove_once:Nn
 128, [12762](#), 12764, 12767
- \tl_greplace_all:Nnn
 128, [12651](#), 12657, 12671, 12673, 12771
- \tl_greplace_once:Nnn
 127, [12651](#), 12653, 12663, 12665, 12765
- \tl_greverse:N 120, [13376](#), 13378, 13381
- .tl_gset:N 250, [23108](#)
- \tl_gset:Nn 114,
 133, 159, 714, 725, 756, 7032, 7041,
 8873, 8887, 8893, 8902, 8903, 8913,
 8914, 8928, 9288, 9290, 9292, 9294,
 9296, [12303](#), 12307, 12309, 12315,
 12316, 12317, 12318, 12481, 12573,
 17538, 17806, 17875, 41255, 41307
- .tl_gset_e:N 251, [23108](#)
- \tl_gset_eq:NN .. 114, 3288, 7038,
 7329, 8389, [12271](#), 12273, 12276,
 12752, 13711, 14629, 14645, 17405,
 17406, 17407, 17408, 19251, 19252,
 19253, 19254, 26467, 42114, 42277
- \tl_gset_rescan:Nnn .. 129, [12469](#),
 12480, 12511, 12512, [12568](#), 12572
- .tl_gset_x:N [41446](#)
- \tl_gsort:Nn
 127, [3286](#), 3288, 3289, [13148](#)
- \tl_gtrim_left_spaces:N
 121, [13059](#), 13085, 13093
- \tl_gtrim_right_spaces:N
 121, [13059](#), 13087, 13094
- \tl_gtrim_spaces:N
 121, [13059](#), 13083, 13092
- \tl_head:N 124, [13148](#), 13161
- \tl_head:n 124, 740, 741,
 749, 12613, [13148](#), 13148, 13158,
 13161, 13419, 26517, 32494, 32503
- \tl_head:w ... 124, 741, 742, [13148](#),
 13159, 32206, 32532, 32561, 39941
- \tl_if_blank:n 12810, 12818
- \tl_if_blank:nTF
 . 115, 124, 3461, 6674, 8872, 9171,
 9173, 9501, 10452, 11011, 11181,
 11203, 11236, 11271, 11313, 11377,
 11384, 11454, 11463, 11487, 11563,
 12599, [12809](#), 13165, 13407, 13694,
 14406, 14409, 15769, 15772, 19682,
 19800, 22920, 23272, 23529, 23630,
 23648, 23649, 23670, 23681, 23707,
 31949, 31952, 31955, 31987, 31990,
 32123, 32208, 32225, 32235, 32248,
 32268, 32309, 32527, 32555, 33388,
 33399, 33613, 33856, 33860, 34528,
 34538, 34649, 34674, 34731, 35121,
 35143, 36082, 36094, 37626, 38792,
 38813, 38822, 38827, 38842, 38947,
 39231, 39442, 39450, 40078, 40081,
 40084, 40108, 40134, 40160, 41427
- \tl_if_blank_p:n ... 115, 11333, [12809](#)
- \tl_if_empty:N 12776,
 12784, 13810, 13812, 19486, 19488
- \tl_if_empty:n
 12786, 12794, 12801, 13814
- \tl_if_empty:NIF
 115, 7060, 9201, 9230, 9400,
 9410, 10822, 10912, 10947, 11028,
 11293, 11420, 11435, [12776](#), 23306,
 23311, 23505, 38764, 39167, 39454,
 39816, 39837, 40502, 40729, 41735
- \tl_if_empty:nTF
 115, 531, 730, 732, 733,
 910, 919, 925, 1681, 1777, 2152,
 4349, 4350, 5830, 7673, 7837, 8322,
 8477, 9658, 9768, 9783, 9876, 9880,
 9938, 10052, 10094, 10097, 10129,
 10157, 10158, 10168, 10175, 10181,
 10188, 10816, 11063, 11591, 11597,
 11599, 11601, 11804, 12677, [12786](#),
 [12796](#), 12864, 12905, 13257, 13520,
 13555, 13768, 14679, 15735, 16706,
 16789, 16863, 16919, 17010, 17084,
 17091, 17108, 17258, 17451, 19190,
 19209, 19218, 19221, 19499, 19532,
 19646, 19735, 20451, 20725, 21525,
 21529, 22621, 22624, 22721, 23762,
 24967, 25823, 26515, 30279, 30321,
 30789, 31499, 32549, 40422, 41901
- \tl_if_empty_p:N

- [115](#), [11725](#), [12776](#), [39221](#)
- \tl_if_empty_p:n ... [115](#), [12786](#), [12796](#)
- \tl_if_eq:NN [12820](#), [12821](#)
- \tl_if_eq:Nn [12825](#), [12837](#)
- \tl_if_eq:nn [12838](#), [12851](#)
- \tl_if_eq:NNTF [115](#), [136](#),
[151](#), [762](#), [863](#), [9749](#), [9803](#), [12820](#),
[13559](#), [17607](#), [32381](#), [38853](#), [39170](#)
- \tl_if_eq:NnTF [115](#), [12825](#)
- \tl_if_eq:nnTF [103](#),
[115](#), [136](#), [162](#), [193](#), [863](#), [12838](#), [19520](#)
- \tl_if_eq_p:NN [115](#), [12820](#)
- \tl_if_exist:N
..... [12295](#), [12296](#), [13806](#), [13808](#)
- \tl_if_exist:NNTF
..... [114](#), [4003](#), [11740](#), [12266](#),
[12268](#), [12295](#), [13009](#), [32654](#), [33432](#),
[33727](#), [33733](#), [34894](#), [39499](#), [39781](#)
- \tl_if_exist_p:N
[114](#), [11724](#), [12295](#), [32949](#), [33412](#), [40876](#)
- \tl_if_head_eq_catcode:nN
..... [742](#), [743](#), [13185](#), [13201](#)
- \tl_if_head_eq_catcode:nNTF
..... [117](#), [13171](#)
- \tl_if_head_eq_catcode_p:nN
..... [117](#), [13171](#)
- \tl_if_head_eq_charcode:nN
..... [741](#), [743](#), [13171](#), [13183](#)
- \tl_if_head_eq_charcode:nNTF ...
..... [117](#), [13171](#), [16908](#), [32237](#), [33798](#)
- \tl_if_head_eq_charcode_p:nN ...
..... [117](#), [13171](#), [36296](#)
- \tl_if_head_eq_meaning:nN
..... [742](#), [13203](#), [13210](#)
- \tl_if_head_eq_meaning:nNTF
..... [117](#), [5917](#), [13171](#)
- \tl_if_head_eq_meaning_p:nN
..... [117](#), [5749](#), [13171](#),
[32662](#), [33108](#), [33109](#), [33110](#), [33111](#)
- \tl_if_head_is_group:n [13264](#)
- \tl_if_head_is_group:nTF ... [117](#),
[13191](#), [13231](#), [13264](#), [13316](#), [19216](#),
[33018](#), [33069](#), [33306](#), [33449](#), [33534](#),
[33664](#), [34949](#), [35003](#), [35538](#), [35597](#)
- \tl_if_head_is_group_p:n . [117](#), [13264](#)
- \tl_if_head_is_N_type:n .. [742](#), [13244](#)
- \tl_if_head_is_N_type:nTF
..... [118](#), [12902](#), [13174](#),
[13188](#), [13205](#), [13244](#), [13480](#), [33015](#),
[33066](#), [33219](#), [33310](#), [33446](#), [33531](#),
[33661](#), [33762](#), [34003](#), [34030](#), [34129](#),
[34218](#), [34466](#), [34498](#), [34561](#), [34684](#),
[34741](#), [34783](#), [34819](#), [34946](#), [35000](#),
[35071](#), [35111](#), [35253](#), [35535](#), [35594](#)
- \tl_if_head_is_N_type_p:n [118](#), [13244](#)
- \tl_if_head_is_space:n [13279](#)
- \tl_if_head_is_space:nTF ... [118](#),
[125](#), [13279](#), [13462](#), [13471](#), [14304](#), [21527](#)
- \tl_if_head_is_space_p:n . [118](#), [13279](#)
- \tl_if_in:Nn [919](#), [12857](#)
- \tl_if_in:nn [12859](#), [12868](#)
- \tl_if_in:NnTF [116](#), [12699](#),
[12854](#), [12854](#), [12855](#), [12856](#), [17346](#)
- \tl_if_in:nnTF
..... [116](#), [732](#), [763](#), [4549](#), [8970](#), [9644](#),
[9646](#), [10339](#), [10589](#), [12518](#), [12683](#),
[12685](#), [12854](#), [12855](#), [12856](#), [12859](#),
[13851](#), [13859](#), [20723](#), [30792](#), [38279](#)
- \tl_if_noalue:n [12872](#)
- \tl_if_noalue:nTF [116](#), [12870](#)
- \tl_if_noalue_p:n [116](#), [12870](#)
- \tl_if_regex_match:nN
..... [12914](#), [12921](#), [12926](#)
- \tl_if_regex_match:nn
..... [12914](#), [12914](#), [12919](#)
- \tl_if_regex_match:nNTF [117](#)
- \tl_if_regex_match:nnTF [117](#)
- \tl_if_single:N [12888](#)
- \tl_if_single:n [12889](#)
- \tl_if_single:NNTF
..... [116](#), [12884](#), [12885](#), [12886](#), [12887](#)
- \tl_if_single:nTF [116](#),
[607](#), [733](#), [5911](#), [5947](#), [5962](#), [5995](#),
[12885](#), [12886](#), [12887](#), [12889](#), [34765](#)
- \tl_if_single_p:N [116](#), [12884](#)
- \tl_if_single_p:n [116](#), [12884](#),
[12889](#), [33817](#), [34699](#), [34756](#), [34838](#)
- \tl_if_single_token:n [12900](#)
- \tl_if_single_token:nTF
..... [116](#), [5053](#), [12900](#), [34904](#)
- \tl_if_single_token_p:n
..... [116](#), [12900](#), [35047](#)
- \tl_item:Nn
[125](#), [983](#), [13382](#), [13403](#), [13404](#), [21573](#)
- \tl_item:nn [125](#), [578](#), [748](#),
[7381](#), [7425](#), [13382](#), [13382](#), [13403](#), [13408](#)
- \tl_log:N
[121](#), [3998](#), [13512](#), [13514](#), [13515](#), [14446](#)
- \tl_log:n [121](#), [418](#),
[419](#), [1142](#), [2282](#), [2298](#), [8441](#), [9826](#),
[10367](#), [10611](#), [13514](#), [13548](#), [13548](#),
[13550](#), [14442](#), [19115](#), [19168](#), [26492](#)
- \tl_lower_case:n [41563](#), [41564](#)
- \tl_lower_case:nn [41563](#), [41567](#)
- \tl_map_break:
..... [63](#), [123](#), [477](#), [492](#), [3984](#),
[3985](#), [12933](#), [12948](#), [12963](#), [12974](#),
[12989](#), [13000](#), [13000](#), [13001](#), [13003](#)

- \tl_map_break:n 123, 3293, 11210, 13000, 13002, 39313
- \tl_map_function:NN 122, 5874, 12928, 12936, 12938, 13024
- \tl_map_function:nN 122, 157, 2151, 5100, 12928, 12928, 12935, 12937, 13019, 17454, 35349, 35366, 35392
- \tl_map_inline:Nn 122, 3293, 12953, 12966, 12968, 14542, 14544, 39309
- \tl_map_inline:nn 122, 123, 154, 492, 3255, 6271, 6721, 8746, 10704, 12953, 12953, 12967, 20681, 20683, 20685, 25876, 29081, 30693, 30701, 32959, 33332, 33335, 35718, 35735, 35799, 41639, 42013, 42075, 42510, 42571, 42619
- \tl_map_tokens:Nn 122, 11209, 12969, 12976, 12978
- \tl_map_tokens:nn 122, 7039, 12969, 12969, 12977, 12994
- \tl_map_variable:NNn 122, 12993, 12997, 12999
- \tl_map_variable:nNn 123, 736, 12993, 12993, 12998
- \tl_mixed_case:n 41563, 41576
- \tl_mixed_case:nn 41563, 41579
- \tl_new:N 113, 114, 206, 716, 3182, 3607, 3626, 4423, 4424, 4430, 4431, 6637, 6642, 6643, 6649, 6650, 6651, 6907, 6909, 7028, 7029, 7844, 7845, 7846, 7847, 8717, 9068, 9244, 9259, 9306, 9694, 9695, 10263, 10266, 10288, 10508, 10519, 10553, 10678, 10680, 10693, 10695, 10696, 10698, 11003, 11033, 11034, 12246, 12246, 12251, 12266, 12268, 12823, 12824, 13576, 13577, 13578, 13579, 14453, 14454, 17379, 17380, 19181, 19233, 19234, 20021, 20539, 20740, 22459, 22465, 22470, 22472, 22480, 22482, 22483, 22485, 31128, 31514, 31515, 32931, 32932, 32933, 32940, 32942, 32944, 32980, 37354, 38403, 38641, 38644, 38741, 38742, 38743, 38744, 39164, 39251, 39373, 39492, 40431, 40433, 41234, 41335, 41725, 41917, 41918, 41919, 42775
- \tl_put_left:Nn 114, 12319, 12319, 12324, 12329, 12334, 12342, 12381, 12382, 12383, 12384, 12385, 12386, 42196, 42197, 42198, 42199, 42200
- \tl_put_right:Nn 115, 132, 755, 4199, 4266, 4276, 4315, 4337, 4659, 10953, 10956, 10961, 12393, 12393, 12395, 12400, 12405, 12413, 12449, 12450, 12451, 12452, 12453, 12454, 17567, 19418, 20036, 20038, 20039, 20040, 20042, 20044, 20046, 20047, 20049, 20051, 20053, 20055, 22496, 32971, 39973, 40019, 41728, 42201, 42202, 42203, 42204, 42205
- \tl_rand_item:N 125, 13405, 13410, 13411
- \tl_rand_item:n 125, 13405, 13405, 13410
- \tl_range:Nnn 126, 13414, 13414, 13415
- \tl_range:nnn 126, 141, 13414, 13414, 13416
- \tl_regex_greplac_all:NNn 128, 12731
- \tl_regex_greplac_all:Nnn 128, 12731
- \tl_regex_greplac_once:NNn 128, 12731
- \tl_regex_greplac_once:Nnn 128, 12731
- \tl_regex_replac_all:NNn 128, 12731, 12740, 12742
- \tl_regex_replac_all:Nnn 128, 12731, 12737, 12739
- \tl_regex_replac_once:NNn 128, 12731, 12734, 12736
- \tl_regex_replac_once:Nnn 128, 12731, 12731, 12733
- \tl_remove_all:Nn 128, 129, 12768, 12768, 12772, 12773
- \tl_remove_once:Nn 128, 12762, 12762, 12766
- \tl_replac_all:Nnn 128, 859, 917, 12651, 12655, 12667, 12669, 12769, 17463
- \tl_replac_once:Nnn 127, 12651, 12651, 12659, 12661, 12763
- \tl_rescan:nn 129, 130, 297, 720, 12469, 12469, 12475
- \tl_retokenize:n 130, 12638, 12640, 12650
- \tl_reverse:N 119, 120, 13376, 13376, 13380
- \tl_reverse:n ... 119, 120, 12613, 13357, 13357, 13369, 13377, 13379
- \tl_reverse_items:n 119, 120, 13043, 13043
- .tl_set:N 250, 23108
- \tl_set:Nn 114, 129, 130, 132, 133, 159, 251, 435, 643, 714, 716, 725, 756, 961, 970, 4125, 5020, 5781, 5858, 6123, 6186, 6713, 6733, 6743, 6759, 6764, 6807, 6839, 6877, 7227, 7898, 7899,

- 7959, 8960, 8995, 9216, 9439, 9660,
 9706, 9789, 10411, 10424, 10457,
 10769, 10806, 11133, 11170, 11286,
 11389, 11392, 11395, 11398, 11427,
 11675, 11678, 11679, 11680, 11681,
 11688, 11689, 11690, 11692, 11696,
12303, 12303, 12305, 12311, 12312,
 12313, 12314, 12479, 12571, 12595,
 12601, 12608, 12828, 12841, 12842,
 12996, 13537, 13558, 14700, 14850,
 17453, 17457, 17536, 17603, 17612,
 17635, 17757, 17760, 17777, 17785,
 17804, 17813, 17872, 18003, 18692,
 19342, 19348, 19357, 19364, 19607,
 20034, 20614, 20779, 20914, 20960,
 21066, 21074, 21193, 21199, 21209,
 21215, 21273, 21492, 21879, 22473,
 22697, 22759, 22906, 23140, 23408,
 23410, 26871, 31648, 31658, 32224,
 32229, 32295, 32364, 32383, 32394,
 32406, 32934, 32941, 32943, 32945,
 32976, 33329, 33896, 33912, 33928,
 35715, 38280, 38281, 38408, 38642,
 38671, 38747, 38774, 38781, 38798,
 38806, 38809, 38859, 38874, 39175,
 39181, 39182, 39186, 39240, 39248,
 39252, 39383, 39391, 39409, 39412,
 39453, 39469, 39493, 39504, 39523,
 39556, 39558, 39560, 39563, 39580,
 39774, 39962, 40445, 40628, 40704,
 40728, 40755, 40758, 40762, 42773
 .tl_set_e:N 251, 23108
 \tl_set_eq:NN
 114, 191, 604, 3286, 6901,
 7324, 7817, 8388, 9752, 9760, 12271,
 12271, 12275, 12749, 13710, 14627,
 14636, 17401, 17402, 17403, 17404,
 19247, 19248, 19249, 19250, 22586,
 22758, 23256, 23286, 23373, 23401,
 23470, 26466, 32297, 32298, 32378,
 38661, 38776, 38873, 39455, 39475,
 39500, 39817, 40637, 42113, 42195
 \tl_set_rescan:Nnn 129, 130,
297, 684, 721, 12469, 12471, 12478,
 12509, 12510, 12568, 12570, 40568
 .tl_set_x:N 41446
 \tl_show:N 121, 191,
478, 3995, 13512, 13512, 13513, 14439
 \tl_show:n
 90, 121, 418, 419, 646, 751,
752, 1142, 2278, 2295, 8439, 9824,
 10365, 10609, 13512, 13532, 13532,
 13534, 14435, 19113, 19167, 19873,
 19943, 19949, 19955, 19961, 26490
 \tl_sort:Nn 127, 3286, 3286, 3287, 13148
 \tl_sort:nN
 127, 456, 457, 3457, 3457, 13148
 \tl_tail:N
 ... 124, 791, 5724, 13148, 13170,
 14845, 21549, 21557, 21567, 41734
 \tl_tail:n
 124, 13148, 13162, 13169, 13170
 \tl_to_str:N
101, 119, 134, 570, 673, 760, 10764,
 10775, 11639, 11653, 13005, 13005,
 13006, 13563, 13564, 13776, 13843,
 13851, 14438, 14445, 15007, 19847
 \tl_to_str:n 54,
56, 79, 101, 118, 119, 129, 130,
134, 143, 144, 219, 221, 222, 246,
388, 400, 570, 730, 733, 760, 767,
773, 943, 980, 1039, 1285, 1412,
 1412, 1435, 1666, 1753, 2334, 2768,
 2782, 2785, 2792, 2796, 3066, 3108,
 3126, 5100, 6057, 7184, 7343, 7419,
 7442, 7516, 8427, 8433, 8965, 9171,
 9177, 9535, 9536, 9672, 9674, 9678,
 9680, 9685, 9687, 10226, 10227,
 10228, 10229, 10334, 10584, 10686,
 10702, 11068, 11186, 11197, 11263,
 12491, 12647, 12680, 12788, 13004,
 13004, 13533, 13549, 13777, 13851,
 13859, 14010, 14032, 14057, 14064,
 14118, 14125, 14199, 14218, 14230,
 14256, 14264, 14272, 14278, 14290,
 14301, 14403, 14414, 14541, 14654,
 14659, 14664, 14726, 15747, 16630,
 16630, 16763, 16813, 16840, 16895,
 17331, 17342, 18969, 18986, 19030,
 19120, 19174, 20240, 20244, 20274,
 20275, 20309, 20324, 20326, 20328,
 20445, 20718, 20723, 20756, 21038,
 21092, 21094, 21111, 21139, 21141,
 21275, 21293, 21361, 21500, 21506,
 21507, 21719, 21919, 22122, 22555,
 22735, 23139, 23592, 23694, 23696,
 23702, 23703, 24183, 24382, 24386,
 24403, 24599, 24600, 25232, 25233,
 25238, 25242, 30035, 30089, 30163,
 30573, 30580, 30581, 30759, 30867,
 30882, 30902, 30923, 30963, 31028,
 31042, 31111, 31202, 31365, 31403,
 31504, 31723, 32774, 33432, 33727,
 33730, 33733, 33736, 33938, 38429,
 38503, 38753, 39426, 39619, 40225,
 41397, 41419, 41422, 41632, 41634,
 41689, 41697, 41749, 41757, 42032,
 42035, 42044, 42045, 42046, 42055,

- [42057](#), [42090](#), [42092](#), [42504](#), [42740](#)
`\tl_trim_left_spaces:N`
 [121](#), [13059](#), [13079](#), [13090](#)
`\tl_trim_left_spaces:n`
 [120](#), [13059](#), [13061](#), [13066](#), [13080](#), [13086](#)
`\tl_trim_left_spaces_apply:nN` ...
 [121](#), [13059](#), [13070](#), [13075](#)
`\tl_trim_right_spaces:N`
 [121](#), [13059](#), [13081](#), [13091](#)
`\tl_trim_right_spaces:n`
 [120](#), [13059](#), [13063](#), [13067](#), [13082](#), [13088](#)
`\tl_trim_right_spaces_apply:nN` ..
 [121](#), [13059](#), [13072](#), [13076](#)
`\tl_trim_spaces:N`
 [121](#), [13059](#), [13077](#), [13089](#)
`\tl_trim_spaces:n` [120](#), [739](#),
 [1044](#), [11116](#), [13059](#), [13059](#), [13065](#),
 [13078](#), [13084](#), [17442](#), [17444](#), [41427](#)
`\tl_trim_spaces_apply:nN` [120](#), [121](#),
 [1013](#), [11113](#), [13059](#), [13068](#), [13074](#),
 [19192](#), [19649](#), [19739](#), [19813](#), [31365](#)
`\tl_upper_case:n` [41563](#), [41570](#)
`\tl_upper_case:nn` [41563](#), [41573](#)
`\tl_use:N`
 [119](#), [197](#), [235](#), [240](#), [243](#), [9063](#),
 [9239](#), [13007](#), [13007](#), [13015](#), [19721](#),
 [22821](#), [31441](#), [32379](#), [32386](#), [32391](#),
 [32395](#), [32398](#), [32399](#), [32407](#), [32424](#),
 [32425](#), [32581](#), [32655](#), [42030](#), [42034](#)
`\g_tmpa_tl` [131](#), [13576](#)
`\l_tmpa_tl` [7](#), [61](#), [129](#), [131](#),
 [1236](#), [1238](#), [1255](#), [1280](#), [1282](#), [1286](#),
 [1288](#), [1292](#), [1294](#), [1298](#), [1300](#), [13578](#)
`\g_tmpb_tl` [131](#), [13576](#)
`\l_tmpb_tl` [131](#), [1237](#),
 [1238](#), [1253](#), [1255](#), [1281](#), [1282](#), [1287](#),
 [1288](#), [1293](#), [1294](#), [1299](#), [1300](#), [13578](#)
- tl internal commands:
- `\__tl_act:NNNn`
 [745–747](#), [13033](#), [13295](#), [13346](#), [13362](#)
`\__tl_act_count_group:n` [13035](#), [13042](#)
`\__tl_act_count_group:nn` [13029](#)
`\__tl_act_count_normal:N` [13034](#), [13040](#)
`\__tl_act_count_normal:nN` [13029](#)
`\__tl_act_count_space:` [13036](#), [13041](#)
`\__tl_act_count_space:n` [13029](#)
`\__tl_act_end:wn`
 [737](#), [13295](#), [13330](#), [13334](#)
`\__tl_act_group:nwNNN`
 [13295](#), [13317](#), [13332](#)
`\__tl_act_if_head_is_space:nTF` ..
 [746](#), [13295](#), [13297](#), [13313](#), [13322](#)
`\__tl_act_if_head_is_space:w` ...
 [13295](#), [13299](#), [13303](#)
- `\__tl_act_if_head_is_space-true:w` [13295](#), [13300](#), [13306](#)
`\__tl_act_loop:w` . [745](#), [746](#), [13295](#),
 [13311](#), [13326](#), [13336](#), [13343](#), [13349](#)
`\__tl_act_normal:NwNNN`
 [13295](#), [13318](#), [13323](#)
`\__tl_act_output:n` . [747](#), [13295](#), [13353](#)
`\__tl_act_result:n` ... [747](#), [13330](#),
 [13351](#), [13353](#), [13354](#), [13355](#), [13356](#)
`\__tl_act_reverse` [747](#)
`\__tl_act_reverse_output:n`
 [13295](#), [13355](#), [13371](#), [13373](#), [13375](#)
`\__tl_act_space:wwNNN`
 [746](#), [13295](#), [13314](#), [13340](#)
`\__tl_analysis:n`
 [467](#), [478](#), [3659](#), [3659](#), [3974](#), [4005](#), [4017](#)
`\__tl_analysis_a:n` .. [3663](#), [3712](#), [3712](#)
`\__tl_analysis_a_bgroup:w`
 [3743](#), [3765](#), [3767](#)
`\__tl_analysis_a_cs:ww`
 [3822](#), [3836](#), [3839](#)
`\__tl_analysis_a_egroup:w`
 [3745](#), [3765](#), [3770](#)
`\__tl_analysis_a_group:nw`
 [3765](#), [3768](#), [3771](#), [3773](#)
`\__tl_analysis_a_group_aux:w` ...
 [3765](#), [3781](#), [3783](#)
`\__tl_analysis_a_group_auxii:w` ..
 [3765](#), [3788](#), [3791](#)
`\__tl_analysis_a_group_test:w` ...
 [3765](#), [3793](#), [3798](#)
`\__tl_analysis_a_loop:w` ... [3719](#),
 [3722](#), [3722](#), [3763](#), [3805](#), [3819](#), [3837](#)
`\__tl_analysis_a_safe:N`
 [3744](#), [3786](#), [3822](#), [3822](#)
`\__tl_analysis_a_space:w`
 [3742](#), [3748](#), [3748](#)
`\__tl_analysis_a_space_test:w` ...
 [470](#), [3748](#), [3750](#), [3755](#)
`\__tl_analysis_a_store:`
 [470](#), [3759](#), [3801](#), [3807](#), [3807](#)
`\__tl_analysis_a_type:w`
 [3723](#), [3724](#), [3724](#)
`\__tl_analysis_b:n` .. [3664](#), [3850](#), [3850](#)
`\__tl_analysis_b_char:Nn`
 [483](#), [3877](#), [3884](#), [3884](#), [4200](#)
`\__tl_analysis_b_char_aux:nww` ...
 [475](#), [3878](#), [3884](#), [3906](#)
`\__tl_analysis_b_cs:Nww`
 [3880](#), [3912](#), [3912](#)
`\__tl_analysis_b_cs_test:ww`
 [3912](#), [3915](#), [3917](#)
`\__tl_analysis_b_loop:w`
 [476](#), [3850](#), [3854](#), [3858](#), [3959](#), [3968](#)

__tl_analysis_b_normal:wwN
 [3863](#), [3868](#), [3870](#), [3934](#)
 __tl_analysis_b_normals:ww
 . . . [475](#), [3860](#), [3863](#), [3863](#), [3909](#), [3920](#)
 __tl_analysis_b_special:w
 [3866](#), [3931](#), [3933](#)
 __tl_analysis_b_special_char:wN
 [3931](#), [3948](#), [3956](#)
 __tl_analysis_b_special_space:w
 [3931](#), [3950](#), [3963](#)
 __tl_analysis_char_arg:Nw
 [4096](#), [4096](#), [4253](#), [4312](#)
 __tl_analysis_char_arg_aux:Nw
 [4096](#), [4099](#), [4102](#)
 \l__tl_analysis_char_token . [464](#),
 [470](#), [471](#), [3605](#), [3752](#), [3757](#), [3795](#), [3800](#)
 __tl_analysis_cs_space_count:NN
 [3639](#), [3639](#), [3836](#), [3915](#)
 __tl_analysis_cs_space_count:w
 [3639](#), [3643](#), [3648](#), [3652](#)
 __tl_analysis_cs_space_count_-
 end:w [3639](#), [3645](#), [3654](#)
 __tl_analysis_disable:n
 . . . [468](#), [3668](#), [3670](#), [3679](#), [3714](#), [3780](#)
 __tl_analysis_disable_char:N
 [3688](#), [3690](#), [3701](#), [3833](#)
 __tl_analysis_extract_charcode:
 [3627](#), [3627](#), [3775](#), [4227](#)
 __tl_analysis_extract_charcode_-
 aux:w [3627](#), [3629](#), [3632](#), [3637](#)
 \l__tl_analysis_index_int
 . [472](#), [473](#), [3623](#), [3717](#), [3720](#), [3758](#),
 [3776](#), [3813](#), [3816](#), [3842](#), [3844](#), [3937](#)
 __tl_analysis_map:Nn [3970](#), [3976](#), [3979](#)
 __tl_analysis_map:NwNw
 [3970](#), [3982](#), [3988](#), [3992](#)
 \l__tl_analysis_nesting_int
 [469](#), [3624](#), [3718](#), [3809](#), [3818](#)
 \l__tl_analysis_normal_int
 [3622](#), [3716](#), [3761](#),
 [3803](#), [3814](#), [3817](#), [3834](#), [3843](#), [3848](#)
 \g__tl_analysis_result_tl
 [477](#), [3626](#), [3852](#), [3983](#), [4022](#)
 __tl_analysis_show:
 [4007](#), [4018](#), [4020](#), [4020](#)
 __tl_analysis_show:Nnnn
 [4011](#), [4012](#), [4014](#), [4015](#)
 __tl_analysis_show:NnnN
 [3994](#), [3995](#), [3998](#), [4001](#)
 __tl_analysis_show_active:n
 [4035](#), [4064](#), [4066](#)
 __tl_analysis_show_cs:n
 [4031](#), [4064](#), [4064](#)
 \c__tl_analysis_show_etc_str
 [479](#), [4084](#), [4086](#), [4345](#)
 __tl_analysis_show_long:nn
 [4064](#), [4065](#), [4067](#), [4068](#)
 __tl_analysis_show_long_-
 aux:nnnn . . . [4064](#), [4070](#), [4075](#), [4090](#)
 __tl_analysis_show_loop:wNw
 [4020](#), [4022](#), [4026](#), [4042](#)
 __tl_analysis_show_normal:n
 [4038](#), [4044](#), [4044](#)
 __tl_analysis_show_value:N
 [4049](#), [4049](#), [4073](#)
 \l__tl_analysis_token
 [464](#), [465](#), [470](#),
 [471](#), [480](#), [3605](#), [3630](#), [3723](#), [3727](#),
 [3730](#), [3733](#), [3781](#), [3785](#), [3800](#), [4098](#),
 [4225](#), [4234](#), [4239](#), [4248](#), [4308](#), [4331](#)
 \l__tl_analysis_type_int . . . [470](#),
 [472](#), [3625](#), [3726](#), [3741](#), [3809](#), [3811](#), [3815](#)
 __tl_build_begin:NN
 . . [13584](#), [13585](#), [13587](#), [13588](#), [13619](#)
 __tl_build_begin:NNN
 [754](#), [13584](#), [13589](#), [13590](#)
 __tl_build_end_loop:NN
 . . [13648](#), [13651](#), [13656](#), [13658](#), [13664](#)
 __tl_build_get:NNN
 . . [13650](#), [13655](#), [13667](#), [13668](#), [13668](#)
 __tl_build_get:w
 [13668](#), [13669](#), [13670](#), [13676](#)
 __tl_build_get_end:w
 [13668](#), [13674](#), [13678](#)
 __tl_build_last:NNn
 [754](#), [755](#), [13596](#), [13599](#),
 [13611](#), [13615](#), [13629](#), [13630](#), [13670](#)
 __tl_build_put:nn
 [755](#), [13599](#), [13626](#), [13628](#), [13643](#)
 __tl_build_put:nw
 [755](#), [13599](#), [13628](#), [13629](#)
 __tl_build_put_left:NNn
 [13631](#), [13632](#), [13635](#), [13637](#)
 __tl_count:n
 [737](#), [13016](#), [13019](#), [13024](#), [13026](#)
 __tl_head_aux:n [13151](#), [13153](#)
 __tl_head_auxi:nw [13148](#)
 __tl_head_auxii:n [13148](#)
 __tl_head_exp_not:w
 [743](#), [13171](#), [13175](#), [13189](#), [13240](#)
 __tl_if_blank_p:NNw [12809](#)
 __tl_if_empty_if:n
 [730](#), [731](#), [849](#), [12796](#), [12796](#), [12803](#),
 [12812](#), [12875](#), [12903](#), [12907](#), [13291](#)
 __tl_if_head_eq_empty_arg:w
 [742](#), [743](#), [13171](#), [13175](#), [13189](#), [13242](#)

- _tl_if_head_eq_meaning_-normal:nN 13206, 13212
- _tl_if_head_eq_meaning_-special:nN 13207, 13221
- _tl_if_head_is_group_fi-false:w 13264, 13270, 13278
- _tl_if_head_is_N_type_auxi:w ..
..... 744, 13244, 13247, 13255
- _tl_if_head_is_N_type_auxii:n ..
..... 13244, 13259, 13262
- _tl_if_head_is_space:w
..... 13279, 13282, 13289
- _tl_if_novalue:w
..... 732, 12870, 12875, 12881
- _tl_if_recursion_tail_break:nN
..... 748, 12466, 12466, 13398
- _tl_if_recursion_tail_stop:nTF
..... 12466
- _tl_if_recursion_tail_stop_p:n
..... 12466
- _tl_if_single:nnw
..... 733, 12889, 12891, 12899
- _tl_item:nn
..... 13382, 13384, 13396, 13401
- _tl_item_aux:nn 13382, 13385, 13390
- _tl_map_function:Nnnnnnnnn ...
735, 12928, 12930, 12939, 12944, 12958
- _tl_map_function_end:w
.... 734, 12928, 12942, 12946, 12950
- _tl_map_tokens:nnnnnnnn ...
..... 12969, 12971, 12979, 12985
- _tl_map_tokens_end:w
..... 12969, 12982, 12987, 12991
- _tl_map_variable:Nnn
..... 12993, 12994, 12995
- _tl_peek_analysis_active_str:n
..... 4103, 4262, 4264
- _tl_peek_analysis_char:N
..... 483, 4103, 4181, 4191
- _tl_peek_analysis_char:w
..... 483, 4103, 4204, 4212
- _tl_peek_analysis_collect:n ...
..... 4103, 4312, 4313
- _tl_peek_analysis_collect:w ...
..... 4103, 4309, 4311, 4332
- _tl_peek_analysis_collect_-end:NNNN 4103, 4329, 4334
- _tl_peek_analysis_collect_-loop: 4103, 4316, 4318
- _tl_peek_analysis_collect_-test: 4103
- _tl_peek_analysis_cs:N
..... 4103, 4183, 4187
- _tl_peek_analysis_escape:
..... 4103, 4260, 4305
- _tl_peek_analysis_exp:N
..... 4103, 4144, 4152
- _tl_peek_analysis_exp_aux:N . 4103
- _tl_peek_analysis_exp_aux:Nw ..
..... 482, 4162, 4167
- _tl_peek_analysis_explicit:n ..
..... 4103, 4259, 4274
- _tl_peek_analysis_loop:NNn ...
..... 4103, 4113, 4120, 4122
- _tl_peek_analysis_nonexp:N ...
..... 4103, 4147, 4175, 4240
- _tl_peek_analysis_retest:
..... 483, 484, 4103, 4235, 4237
- _tl_peek_analysis_special: ...
..... 4103, 4149, 4223
- _tl_peek_analysis_str:
..... 4103, 4242, 4245
- _tl_peek_analysis_str:n
..... 4103, 4253, 4254
- _tl_peek_analysis_str:w
..... 4103, 4249, 4252
- _tl_peek_analysis_test:
..... 481, 4103, 4131, 4133
- \c_tl_peek_catcodes_tl 3608
- \l_tl_peek_charcode_int
..... 4095, 4226, 4228, 4231, 4258
- \l_tl_peek_code_tl
..... 482, 3607, 4125, 4154,
4156, 4157, 4165, 4188, 4199, 4208,
4266, 4272, 4276, 4303, 4337, 4343
- _tl_quark_if_nil:n 12467
- _tl_quark_if_nil:nTF 12688
- _tl_range:Nnnn . 13414, 13416, 13417
- _tl_range:nnNn . 13414, 13427, 13437
- _tl_range:nnnNn 13414, 13421, 13425
- _tl_range:w 749, 13414, 13416, 13455
- _tl_range_braced:w 749
- _tl_range_collect:nn ... 13414,
13457, 13466, 13473, 13478, 13492
- _tl_range_collect_braced:w .. 749
- _tl_range_collect_group:nN . 13414
- _tl_range_collect_group:nn ...
..... 13482, 13491
- _tl_range_collect_N:nN
..... 13414, 13481, 13490
- _tl_range_collect_space:nw ...
..... 13414, 13474, 13489
- _tl_range_items:nnNn 749
- _tl_range_normalize:nn
..... 13429, 13433, 13493, 13493
- _tl_range_skip:w
.... 749, 13414, 13444, 13446, 13449


```

\__tl_range_skip_spaces:n .....
..... 13414, 13458, 13460, 13463
\__tl_replace:NnNNNnn .....
..... 725, 726, 12652, 12654,
12656, 12658, 12675, 12675, 12686
\__tl_replace_auxi:NnnNNNnn ....
..... 726, 12675, 12689, 12690, 12697, 12700
\__tl_replace_auxii:nNNNnn .....
..... 726, 727, 12675, 12693, 12701, 12703
\__tl_replace_next:w .....
..... 725, 727, 728, 12656,
12658, 12675, 12708, 12728, 12730
\__tl_replace_next_aux:w .....
..... 12675, 12717, 12728
\__tl_replace_wrap:w .....
..... 725, 727, 728, 12652,
12654, 12675, 12706, 12710, 12729
\__tl_rescan:NNw .....
..... 720, 12469, 12497, 12504, 12554, 12559
\__tl_rescan_aux: 12469, 12472, 12476
\c__tl_rescan_marker_tl 722, 12468,
12496, 12504, 12534, 12566, 12610
\__tl_reverse_group_preserve:n ..
..... 13364, 13372
\__tl_reverse_group_preserve:nn .
..... 13357
\__tl_reverse_items:nwNwn .....
.. 13043, 13045, 13046, 13050, 13053
\__tl_reverse_items:wn .....
..... 13043, 13047, 13054, 13057
\__tl_reverse_normal:N . 13363, 13370
\__tl_reverse_normal:nN ..... 13357
\__tl_reverse_space: .. 13365, 13374
\__tl_reverse_space:n ..... 13357
\__tl_sep: ..... 466, 749,
3646, 3654, 3656, 3657, 3839, 3854,
3858, 3861, 3863, 3868, 3870, 3882,
3918, 3928, 3929, 3934, 3952, 3956,
3959, 3964, 3968, 13413, 13413,
13443, 13444, 13446, 13450, 13455
\__tl_set_rescan:nNN .....
..... 720, 722, 12491, 12513, 12513
\__tl_set_rescan:NNnn 720, 12469,
12479, 12481, 12482, 12571, 12573
\__tl_set_rescan:w .... 12596, 12603
\__tl_set_rescan_aux:w .....
.. 12568, 12591, 12600, 12607, 12610
\__tl_set_rescan_multi:nNN .....
..... 720, 722, 12469, 12494,
12521, 12543, 12568, 12588, 12611
\__tl_set_rescan_multi_aux:nNN ..
..... 12568, 12615, 12617, 12622

\__tl_set_rescan_single:nnNN ...
..... 722, 12513, 12524,
12528, 12540, 12568, 12574, 12585
\__tl_set_rescan_single:NNww .. 722
\__tl_set_rescan_single_aux:nnNN
..... 12568, 12579, 12593
\__tl_set_rescan_single_aux:nnnNN
..... 12513, 12533, 12546
\__tl_set_rescan_single_aux:w ...
..... 722, 12513, 12551, 12565
\__tl_show:n .... 13532, 13533, 13535
\__tl_show:NN .....
..... 13512, 13512, 13514, 13516
\__tl_show:w .... 13532, 13537, 13547
\__tl_tl_head:w . 13148, 13160, 13215
\__tl_tmp:w ..... 732,
739, 12744, 12757, 12758, 12759,
12760, 12863, 12864, 12870, 12883,
13095, 13147, 13295, 13310, 13580
\l__tl_tmpa_tl .....
.... 751, 4307, 4315, 4322, 4323,
4339, 12471, 12473, 12477, 12595,
12601, 12603, 12605, 12608, 12749,
12751, 12752, 12823, 12841, 12845,
13537, 13543, 13558, 13559, 13564
\l__tl_tmppb_tl .....
.. 12823, 12828, 12831, 12842, 12845
\__tl_trim_left_spaces:nn .....
.... 739, 13062, 13071, 13095, 13106
\__tl_trim_mark: .....
.. 738, 739, 13095, 13100, 13101,
13102, 13103, 13109, 13110, 13111,
13112, 13118, 13123, 13124, 13126,
13127, 13130, 13142, 13143, 13145
\__tl_trim_right_spaces:nn .....
.... 739, 13064, 13073, 13095, 13115
\__tl_trim_spaces:nn .....
..... 739, 1013, 13060, 13069, 13095, 13097
\__tl_trim_spaces_auxi:w .....
..... 739, 13095,
13099, 13101, 13108, 13110, 13122,
13124, 13126, 13127, 13142, 13143
\__tl_trim_spaces_auxii:w .....
..... 739, 13095, 13103, 13125
\__tl_trim_spaces_auxiii:w .....
..... 739, 13095, 13117, 13118,
13129, 13130, 13133, 13134, 13137
\__tl_trim_spaces_auxiv:w .....
.... 739, 13095, 13119, 13131, 13135
\__tl_trim_spaces_auxv:w .....
..... 739, 13095, 13112, 13140
\__tl_use_none_delimit_by_q_act_-
stop:w ..... 13295

```

- _tl_use_none_delimit_by_s_act_-
stop:w 13329, 13334
- _tl_use_none_delimit_by_s_-\br/>stop:w 734, 12928,
12941, 12948, 12952, 12981, 12989
- \tojis 1192
- token commands:
- \c_alignment_token
..... 206, 939, 1498, 3894,
20106, 20141, 20180, 32781, 39277
- \c_catcode_letter_token 206,
940, 3890, 20097, 20141, 20209, 32793
- \c_catcode_other_token 206,
940, 3888, 20100, 20141, 20214, 32796
- \c_group_begin_token 206
- \c_group_end_token 206
- \c_math_subscript_token
..... 206, 940, 3898, 20115,
20141, 20199, 32752, 32787, 39292
- \c_math_superscript_token
..... 206, 939, 3896,
20112, 20141, 20194, 32784, 39293
- \c_math_toggle_token
..... 206, 939, 3892,
20103, 20141, 20175, 32749, 32778
- \c_parameter_token
.. 206, 563, 939, 20141, 20184, 20187
- \c_space_token
..... 42, 118, 131, 206, 213,
742, 940, 3727, 3757, 3900, 4098,
4138, 4278, 4727, 4768, 7077, 10877,
13193, 13233, 20124, 20141, 20204,
20565, 20590, 20703, 32753, 32790
- \token_case_catcode:Nn
..... 211, 20496, 20496
- \token_case_catcode:NnTF
211, 20496, 20498, 20500, 20502, 39290
- \token_case_charcode:Nn
..... 211, 20496, 20504
- \token_case_charcode:NnTF
.... 211, 20496, 20506, 20508, 20510
- \token_case_meaning:Nn
.... 211, 20496, 20512, 41581, 41582
- \token_case_meaning:NnTF
..... 211, 5948,
5963, 5996, 6006, 6017, 8447, 20496,
20514, 20516, 20518, 39297, 41583,
41584, 41585, 41586, 41587, 41588
- \token_if_active:N 20217
- \token_if_active:NTF 208, 20217
- \token_if_active_p:N
..... 208, 20217, 33102,
33277, 33776, 33818, 35048, 35651
- \token_if_alignment:N 20178
- \token_if_alignment:NTF .. 207, 20178
- \token_if_alignment_p:N .. 207, 20178
- \token_if_chardef:NTF 209, 4053, 20287
- \token_if_chardef_p:N
..... 209, 20287, 32715
- \token_if_control_symbol:N ... 20351
- \token_if_control_symbol:NTF ...
..... 209, 20351
- \token_if_control_symbol_p:N ...
..... 209, 20351
- \token_if_control_word:N 20336
- \token_if_control_word:NTF 209, 20336
- \token_if_control_word_p:N 209, 20336
- \token_if_cs:N 20251
- \token_if_cs:NTF
..... 208, 20127, 20251, 33098,
33565, 34013, 34040, 34139, 34225,
34475, 35033, 35123, 35268, 35672
- \token_if_cs_p:N
..... 208, 20251, 33276, 34700,
34757, 34791, 34839, 35087, 35650
- \token_if_dim_register:NTF
..... 210, 4055, 20287
- \token_if_dim_register_p:N 210, 20287
- \token_if_eq_catcode:NN 20224
- \token_if_eq_catcode:NNTF
..... 208, 211,
212, 20224, 20497, 20499, 20501, 20503
- \token_if_eq_catcode_p:NN 208, 20224
- \token_if_eq_charcode:NN 20229
- \token_if_eq_charcode:NNTF
.. 208, 211, 212, 4768, 4773, 5409,
5609, 5622, 5624, 5662, 5800, 7063,
7077, 10000, 10877, 16688, 16772,
16779, 16783, 16792, 16794, 16849,
16905, 20229, 20505, 20507, 20509,
20511, 21961, 21981, 21984, 28600
- \token_if_eq_charcode_p:NN 208, 20229
- \token_if_eq_meaning:NN 20222
- \token_if_eq_meaning:NNTF .. 208,
211, 212, 5107, 5114, 5415, 5448,
5597, 5620, 5652, 5787, 5795, 5798,
7132, 7138, 7165, 7172, 7190, 7213,
7230, 10929, 16700, 16921, 20222,
20513, 20515, 20517, 20519, 24726,
25751, 25810, 26543, 26815, 26817,
26822, 26890, 27078, 29201, 30597,
30620, 30745, 30820, 33054, 33079,
33081, 33252, 33290, 33519, 33545,
34988, 35016, 35566, 35607, 39311
- \token_if_eq_meaning_p:NN
..... 208, 20222, 32816
- \token_if_expandable:N 20256

- \token_if_expandable:NTF
..... [208](#), [4051](#), [20256](#), [32810](#)
- \token_if_expandable_p:N . [208](#), [20256](#)
- \token_if_font_selection:NTF ...
..... [210](#), [20287](#)
- \token_if_font_selection_p:N ...
..... [210](#), [20287](#)
- \token_if_group_begin:N [20163](#)
- \token_if_group_begin:NTF [207](#), [20163](#)
- \token_if_group_begin_p:N
..... [207](#), [20163](#), [35659](#)
- \token_if_group_end:N [20168](#)
- \token_if_group_end:NTF .. [207](#), [20168](#)
- \token_if_group_end_p:N
..... [207](#), [20168](#), [35660](#)
- \token_if_int_register:NTF
..... [210](#), [4056](#), [20287](#)
- \token_if_int_register_p:N [210](#), [20287](#)
- \token_if_letter:N [942](#), [20207](#)
- \token_if_letter:NTF [208](#), [20207](#)
- \token_if_letter_p:N [208](#), [20207](#), [33773](#)
- \token_if_long_macro:NTF . [209](#), [20287](#)
- \token_if_long_macro_p:N . [209](#), [20287](#)
- \token_if_macro:N [20236](#)
- \token_if_macro:NTF [208](#), [2342](#), [2350](#),
[2362](#), [2370](#), [2382](#), [2390](#), [20234](#), [20440](#)
- \token_if_macro_p:N [208](#), [20234](#)
- \token_if_math_subscript:N ... [20197](#)
- \token_if_math_subscript:NTF ...
..... [207](#), [20197](#)
- \token_if_math_subscript_p:N ...
..... [207](#), [20197](#)
- \token_if_math_superscript:N . [20191](#)
- \token_if_math_superscript:NTF ..
..... [207](#), [20191](#)
- \token_if_math_superscript_p:N ..
..... [207](#), [20191](#)
- \token_if_math_toggle:N [20173](#)
- \token_if_math_toggle:NTF [207](#), [20173](#)
- \token_if_math_toggle_p:N [207](#), [20173](#)
- \token_if_mathchardef:NTF
..... [210](#), [4054](#), [20287](#)
- \token_if_mathchardef_p:N
..... [210](#), [20287](#), [32716](#)
- \token_if_muskip_register:NTF ...
..... [210](#), [20287](#)
- \token_if_muskip_register_p:N ...
..... [210](#), [20287](#)
- \token_if_other:N [20212](#)
- \token_if_other:NTF [208](#), [20212](#)
- \token_if_other_p:N [208](#), [20212](#)
- \token_if_parameter:N [20185](#)
- \token_if_parameter:NTF .. [207](#), [20183](#)
- \token_if_parameter_p:N .. [207](#), [20183](#)
- \token_if_primitive:N . [20428](#), [20437](#)
- \token_if_primitive:NTF .. [210](#), [20365](#)
- \token_if_primitive_p:N .. [210](#), [20365](#)
- \token_if_protected_long_-
macro:NTF [209](#), [20287](#)
- \token_if_protected_long_macro_-
p:N [209](#), [20287](#), [32815](#), [32964](#), [33107](#)
- \token_if_protected_macro:NTF ...
..... [209](#), [20287](#)
- \token_if_protected_macro_p:N ...
..... [209](#), [20287](#), [32814](#), [32963](#), [33106](#)
- \token_if_skip_register:NTF
..... [210](#), [4057](#), [20287](#)
- \token_if_skip_register_p:N
..... [210](#), [20287](#)
- \token_if_space:N [20202](#)
- \token_if_space:NTF [207](#), [20202](#)
- \token_if_space_p:N [207](#), [20202](#)
- \token_if_toks_register:NTF
..... [210](#), [4058](#), [20287](#)
- \token_if_toks_register_p:N
..... [210](#), [20287](#)
- \token_to_catcode:N [206](#), [20093](#), [20093](#)
- \token_to_meaning:N
..... [22](#), [206](#), [217](#), [941](#), [945](#),
[1410](#), [1410](#), [1426](#), [1437](#), [1978](#), [2353](#),
[2373](#), [2393](#), [2782](#), [3630](#), [4047](#), [4072](#),
[5056](#), [8454](#), [13528](#), [13569](#), [20093](#),
[20240](#), [20308](#), [20444](#), [20706](#), [32758](#)
- \token_to_str:N
..... [7](#), [23](#), [101](#), [134](#), [206](#), [217](#),
[405](#), [482](#), [484](#), [485](#), [682](#), [744](#), [888](#),
[943](#), [1103](#), [1105](#), [1412](#), [1413](#), [1426](#),
[1426](#), [1644](#), [1653](#), [1685](#), [1708](#), [1761](#),
[1766](#), [1781](#), [1802](#), [1803](#), [1823](#), [1978](#),
[2119](#), [2155](#), [2162](#), [2274](#), [2294](#), [2307](#),
[2802](#), [2887](#), [2902](#), [2917](#), [2924](#), [2950](#),
[2959](#), [2996](#), [3062](#), [3083](#), [3103](#), [3554](#),
[3570](#), [3616](#), [3617](#), [3618](#), [3619](#), [3644](#),
[3753](#), [3796](#), [3826](#), [3875](#), [3887](#), [3889](#),
[3891](#), [3901](#), [3942](#), [3953](#), [4007](#), [4046](#),
[4071](#), [4163](#), [4170](#), [4179](#), [4250](#), [4270](#),
[4279](#), [4346](#), [4670](#), [4677](#), [4787](#), [4791](#),
[5527](#), [7058](#), [7355](#), [7418](#), [7441](#), [7515](#),
[8140](#), [8344](#), [8346](#), [8449](#), [8450](#), [8454](#),
[8512](#), [8956](#), [9076](#), [10376](#), [10378](#),
[10620](#), [10622](#), [10748](#), [10749](#), [10750](#),
[10751](#), [10752](#), [10759](#), [11065](#), [11082](#),
[11129](#), [12300](#), [12468](#), [13248](#), [13268](#),
[13524](#), [13528](#), [13563](#), [13569](#), [13970](#),
[14515](#), [14523](#), [14526](#), [17163](#), [17187](#),
[17191](#), [17206](#), [17349](#), [17657](#), [17722](#),
[18181](#), [18452](#), [19841](#), [19847](#), [20093](#),
[20322](#), [20323](#), [20328](#), [20330](#), [20331](#),

- 20332, 20333, 20334, 20338, 20341,
 20357, 20935, 20953, 21513, 21523,
 21542, 21543, 21550, 21551, 21552,
 23319, 23324, 23885, 23933, 24052,
 24094, 24204, 24381, 24396, 24605,
 24606, 25116, 25117, 25146, 25313,
 25364, 25396, 25416, 25431, 25443,
 25444, 25457, 25458, 25484, 25493,
 25495, 25520, 25523, 25548, 25550,
 25564, 25580, 25598, 25668, 25678,
 25679, 25694, 25695, 26026, 26068,
 26260, 26498, 30571, 31176, 31402,
 31468, 31485, 31718, 31763, 31769,
 32732, 32733, 33273, 33281, 33328,
 33329, 33627, 33630, 33692, 33695,
 33704, 33895, 33896, 34871, 34873,
 34891, 34892, 34907, 34910, 34914,
 34917, 35647, 35655, 35688, 35691,
 35714, 35715, 35789, 35792, 35796,
 35805, 35832, 36532, 37402, 37629,
 38573, 41367, 41368, 41396, 41419,
 41422, 41802, 41811, 42362, 42372
- token internal commands:
- \c_token_A_int 20434, 20471
 - \c_token_active_tl 940, 20159, 20219
 - __token_case:NNnTF
 20496, 20497, 20499, 20501,
 20503, 20505, 20507, 20509, 20511,
 20513, 20515, 20517, 20519, 20520
 - __token_case:NNw
 20496, 20522, 20527, 20531
 - __token_case_end:nw
 20496, 20530, 20533
 - __token_delimit_by_ufont:w .. 20268
 - __token_delimit_by_char":w .. 20268
 - __token_delimit_by_count:w .. 20268
 - __token_delimit_by_dimen:w .. 20268
 - __token_delimit_by_macro:w .. 20268
 - __token_delimit_by_muskip:w .. 20268
 - __token_delimit_by_skip:w ... 20268
 - __token_delimit_by_toks:w ... 20268
 - __token_if_control_symbol_-
 false:w 20351, 20355, 20363
 - __token_if_control_word_aux:w 20336
 - __token_if_control_word_false:w
 20339, 20342, 20350
 - __token_if_macro_p:w
 20234, 20239, 20243
 - __token_if_primitive:NNw
 20365, 20443, 20448
 - __token_if_primitive:Nw
 20365, 20472, 20478
 - __token_if_primitive_loop:N ...
 20365, 20454, 20469, 20475
 - __token_if_primitive_lua:N
 20365, 20430
 - __token_if_primitive_nullfont:N
 20365, 20457, 20461
 - __token_if_primitive_space:w ...
 20365, 20452, 20460
 - __token_if_primitive_undefined:N
 20365, 20481, 20487
 - __token_tmp:w 943, 20269,
 20278, 20279, 20280, 20281, 20282,
 20283, 20284, 20285, 20288, 20322,
 20323, 20324, 20325, 20327, 20329,
 20330, 20331, 20332, 20333, 20334
 - __token_to_catcode:N
 20093, 20094, 20095
 - \toks 424, 20334
 - \toksapp 924
 - \toksdef 425, 3550
 - \tokspre 925
 - \tolerance 426
 - \topmark 427
 - \topmarks 529
 - \topskip 428
 - \toucs 1193
 - \tpack 926
 - \tracingassigns 530
 - \tracingcommands 429
 - \tracingfonts 961
 - \tracinggroups 531
 - \tracingifs 532
 - \tracinglostchars 430
 - \tracingmacros 431
 - \tracingnesting 533
 - \tracingonline 432
 - \tracingoutput 433
 - \tracingpages 434
 - \tracingparagraphs 435
 - \tracingrestores 436
 - \tracingscantokens 534
 - \tracingstacklevels 1218
 - \tracingstats 437
 - true 287
 - trunc 283
- try commands:
- try_require 12105
 - \ttfamily 40667
 - type 339
- U
- \u 33333,
 35799, 35815, 35896, 35897, 35912,
 35913, 35922, 35923, 35936, 35937,
 35938, 35964, 35965, 35990, 35991
 - \uccode 438

<code>\Uchar</code>	963	<code>\Umathlimitabovekern</code>	1024
<code>\Ucharcat</code>	964	<code>\Umathlimitabovevgap</code>	1025
<code>\uchyph</code>	439	<code>\Umathlimitbelowbgap</code>	1026
<code>\ucs</code>	1194	<code>\Umathlimitbelowkern</code>	1027
<code>\Udelcode</code>	965	<code>\Umathlimitbelowvgap</code>	1028
<code>\Udelcodenum</code>	966	<code>\Umathnolimitsubfactor</code>	1029
<code>\Udelimiter</code>	967	<code>\Umathnolimitsupfactor</code>	1030
<code>\Udelimiterover</code>	968	<code>\Umathopbinspacing</code>	1031
<code>\Udelimiterunder</code>	969	<code>\Umathopcloseespacing</code>	1032
<code>\Uhexensible</code>	970	<code>\Umathopenbinspacing</code>	1033
<code>\Uleft</code>	971	<code>\Umathopencloseespacing</code>	1034
<code>\Umathaccent</code>	972	<code>\Umathopeninnerspacing</code>	1035
<code>\Umathaxis</code>	973	<code>\Umathopenopenspacing</code>	1036
<code>\Umathbinbinspacing</code>	974	<code>\Umathopenopspacing</code>	1037
<code>\Umathbincloseespacing</code>	975	<code>\Umathopenordspacing</code>	1038
<code>\Umathbininnerspacing</code>	976	<code>\Umathopenpunctspacing</code>	1039
<code>\Umathbinopenspacing</code>	977	<code>\Umathopenrelspacing</code>	1040
<code>\Umathbinopspacing</code>	978	<code>\Umathoperatorsize</code>	1041
<code>\Umathbinordspacing</code>	979	<code>\Umathopinnerspacing</code>	1042
<code>\Umathbinpunctspacing</code>	980	<code>\Umathopopenspacing</code>	1043
<code>\Umathbinrelspacing</code>	981	<code>\Umathopopspacing</code>	1044
<code>\Umathchar</code>	982	<code>\Umathopordspacing</code>	1045
<code>\Umathcharclass</code>	983	<code>\Umathoppunctspacing</code>	1046
<code>\Umathchardef</code>	984	<code>\Umathoprelspacing</code>	1047
<code>\Umathcharfam</code>	985	<code>\Umathordbinspacing</code>	1048
<code>\Umathcharnum</code>	986	<code>\Umathordcloseespacing</code>	1049
<code>\Umathcharnumdef</code>	987	<code>\Umathordinnerspacing</code>	1050
<code>\Umathcharslot</code>	988	<code>\Umathordopenspacing</code>	1051
<code>\Umathclosebinspacing</code>	989	<code>\Umathordopspacing</code>	1052
<code>\Umathclosecloseespacing</code>	990	<code>\Umathordordspacing</code>	1053
<code>\Umathcloseinnerspacing</code>	992	<code>\Umathordpunctspacing</code>	1054
<code>\Umathcloseopenspacing</code>	994	<code>\Umathordrelspacing</code>	1055
<code>\Umathcloseopspacing</code>	995	<code>\Umathoverbarkern</code>	1056
<code>\Umathcloseordspacing</code>	996	<code>\Umathoverbarrule</code>	1057
<code>\Umathclosepunctspacing</code>	997	<code>\Umathoverbarvgap</code>	1058
<code>\Umathcloserelspacing</code>	999	<code>\Umathoverdelimiterbgap</code>	1059
<code>\Umathcode</code>	1000	<code>\Umathoverdelimitervgap</code>	1061
<code>\Umathcodenum</code>	1001	<code>\Umathpunctbinspacing</code>	1063
<code>\Umathconnectoroverlapmin</code>	1002	<code>\Umathpunctcloseespacing</code>	1064
<code>\Umathfractiondelsize</code>	1004	<code>\Umathpunctinnerspacing</code>	1066
<code>\Umathfractiondenomdown</code>	1005	<code>\Umathpunctopenspacing</code>	1068
<code>\Umathfractiondenomvgap</code>	1007	<code>\Umathpunctopspacing</code>	1069
<code>\Umathfractionnumup</code>	1009	<code>\Umathpunctordspacing</code>	1070
<code>\Umathfractionnumvgap</code>	1010	<code>\Umathpunctpunctspacing</code>	1071
<code>\Umathfractionrule</code>	1011	<code>\Umathpunctrelspacing</code>	1073
<code>\Umathinnerbinspacing</code>	1012	<code>\Umathquad</code>	1074
<code>\Umathinnercloseespacing</code>	1013	<code>\Umathradicaldegreeafter</code>	1075
<code>\Umathinnerinnerspacing</code>	1015	<code>\Umathradicaldegreebefore</code>	1077
<code>\Umathinneropenspacing</code>	1017	<code>\Umathradicaldegreeraise</code>	1079
<code>\Umathinneropspacing</code>	1018	<code>\Umathradicalkern</code>	1081
<code>\Umathinnerordspacing</code>	1019	<code>\Umathradicalrule</code>	1082
<code>\Umathinnerpunctspacing</code>	1020	<code>\Umathradicalvgap</code>	1083
<code>\Umathinnerrelspacing</code>	1022	<code>\Umathrelbinspacing</code>	1084
<code>\Umathlimitabovebgap</code>	1023	<code>\Umathrelcloseespacing</code>	1085

<code>\Umathrelinnerspacing</code>	1086	5826, 7069, 7240, 8488, 8510, 9396,
<code>\Umathrelopenspacing</code>	1087	9406, 9409, 9594, 9626, 9632, 9639,
<code>\Umathrelopenspacing</code>	1088	9711, 10834, 11314, 11419, 18450,
<code>\Umathrelordspacing</code>	1089	18493, 18495, 18497, 18499, 18501,
<code>\Umathrelpunctspacing</code>	1090	18503, 18505, 18507, 18509, 18511,
<code>\Umathrelrelspacing</code>	1091	18513, 18515, 18926, 18936, 19041,
<code>\Umathskewedfractionhgap</code>	1092	19045, 19047, 19049, 19050, 19054,
<code>\Umathskewedfractionvgap</code>	1094	21685, 21687, 21689, 21695, 21700,
<code>\Umathspaceafterscript</code>	1096	21705, 21722, 22599, 22600, 22606,
<code>\Umathstackdenomdown</code>	1097	22927, 30816, 32198, 32218, 32265,
<code>\Umathstacknumup</code>	1098	32336, 32584, 33450, 33569, 33640,
<code>\Umathstackvgap</code>	1099	33641, 33696, 33706, 33713, 33722,
<code>\Umathsubshiftdown</code>	1100	33821, 33953, 34087, 34592, 34605,
<code>\Umathsubshiftdrop</code>	1101	34619, 34629, 34743, 34760, 34761,
<code>\Umathsubsupshiftdown</code>	1102	34785, 34799, 34801, 34859, 35099,
<code>\Umathsubsupvgap</code>	1103	35298, 35393, 35731, 38640, 38676,
<code>\Umathsubtopmax</code>	1104	38679, 38680, 38685, 38687, 38691,
<code>\Umathsubbottommin</code>	1105	38692, 38903, 38984, 39241, 39370,
<code>\Umathsupshiftdrop</code>	1106	39632, 39657, 39713, 40002, 40035,
<code>\Umathsupshiftup</code>	1107	40214, 40640, 40681, 41072, 41877
<code>\Umathsupsubbottommax</code>	1108	<code>\use:n</code> 25, 27, 113, 216, 394,
<code>\Umathunderbarkern</code>	1109	441, 447, 458, 555, 592, 593, 654,
<code>\Umathunderbarrule</code>	1110	720, 873, 959, 1042, 1103, 1425,
<code>\Umathunderbarvgap</code>	1111	1487, 1487, 1493, 1493, 1495, 1495,
<code>\Umathunderdelimiterbgap</code>	1112	1609, 1630, 1656, 1716, 1725, 1736,
<code>\Umathunderdelimitervgap</code>	1114	1737, 1747, 2139, 2315, 2331, 2632,
<code>\Umiddle</code>	1116	2763, 2812, 2949, 2980, 3052, 3961,
undefine commands:		4061, 4725, 4750, 4963, 5098, 5462,
<code>.undefine:</code>	251, 23124	5626, 6071, 6137, 6205, 6654, 6709,
<code>\underline</code>	440	6765, 6825, 7021, 7975, 8037, 8722,
<code>\unexpanded</code>	535	8729, 9532, 10215, 10427, 10647,
<code>\unhbox</code>	441	11753, 12758, 12760, 12800, 12809,
<code>\unhcopy</code>	442	12983, 12984, 12987, 13162, 13255,
<code>\uniformdeviate</code>	962	13289, 13311, 13773, 13850, 13858,
<code>\unkern</code>	443	13954, 13975, 13989, 14398, 14772,
<code>\unless</code>	536	16870, 16871, 16879, 17529, 17531,
<code>\Unosubscript</code>	1117	17533, 17993, 17994, 17995, 17996,
<code>\Unosuperscript</code>	1118	19627, 19628, 19631, 20039, 20155,
<code>\unpenalty</code>	444	20234, 20271, 20290, 20435, 20759,
<code>\unskip</code>	445	21431, 21432, 21433, 21434, 21520,
<code>\unvbox</code>	446	21916, 22497, 22510, 22615, 23134,
<code>\unvcopy</code>	447	23321, 23487, 23521, 23542, 23699,
<code>\Uoverdelimiter</code>	1119	23711, 24643, 24651, 24662, 24679,
<code>\uppercase</code>	448	24687, 24719, 25197, 26806, 30577,
<code>\uptexrevision</code>	1205	30763, 30771, 30780, 31071, 31076,
<code>\uptexversion</code>	1206	31720, 31821, 31939, 31971, 32254,
<code>\Uradical</code>	1120	32541, 32542, 32544, 32836, 32840,
<code>\Uright</code>	1121	33471, 33554, 33847, 34961, 35025,
<code>\Uroot</code>	1122	35741, 36335, 36527, 37784, 37844,
usage commands:		38309, 38316, 38819, 38973, 39209,
<code>.usage:n</code>	254, 23126	39435, 39981, 40096, 40149, 40553,
use commands:		40573, 40741, 40781, 41018, 41033,
<code>\use:N</code>	22, 23, 187,	41646, 41729, 41944, 42041, 42081
401, 1486, 1486, 1680, 1776, 4994,		<code>\use:nn</code> 25, 1495, 1496, 2465,

- 4029, 8990, 9677, 9679, 9684, 9686,
11165, 12503, 12564, 13999, 14571,
16628, 16628, 16712, 16751, 16832,
17005, 21718, 25227, 25236, 25240,
28782, 31221, 32366, 32705, 39761
\use:nnn 25,
1495, 1497, 2116, 16628, 16629, 16728
\use:nnnn 25, 1495, 1498
\use_i:nn ... 26, 393, 399–401, 863,
864, 960, 1232, 1235, 1249, 1253,
1254, 1430, 1499, 1499, 1590, 1674,
1696, 1738, 1946, 2095, 2983, 3040,
3378, 3433, 3443, 3453, 3828, 4802,
4943, 5249, 5255, 5832, 6777, 8724,
12567, 15073, 15078, 15159, 15163,
16816, 17536, 17538, 17622, 17624,
17980, 18032, 18092, 18176, 20799,
21146, 21373, 24046, 24048, 24362,
25007, 25197, 26643, 26977, 27276,
27771, 28065, 28609, 28777, 29091,
29101, 29105, 29613, 29846, 30384,
30409, 30640, 38657, 39315, 41855
\use_i:nnn
.... 26, 797, 963, 968, 977, 1501,
1501, 2353, 3074, 4723, 4748, 4960,
7142, 14557, 15081, 15588, 17845,
20903, 21045, 21380, 21418, 22622,
25166, 27232, 28750, 31155, 31367
\use_i:nnnn 26, 382, 608,
609, 1501, 1504, 1827, 2064, 8482,
8484, 8499, 8504, 8520, 8522, 22504,
26803, 27251, 27258, 27456, 30395
\use_i:nnnnn 26, 1501, 1508
\use_i:nnnnnn 26, 1501, 1513
\use_i:nnnnnnn 26, 1501, 1519
\use_i:nnnnnnnn 26, 1501, 1526
\use_i:nnnnnnnnn 26, 1501, 1534
\use_i_delimit_by_q_nil:nw
..... 28, 1548, 1548
\use_i_delimit_by_q_recursion_-
stop:nw 28,
1548, 1550, 17077, 17093, 33414, 39113
\use_i_delimit_by_q_recursion_-
stop:w 153
\use_i_delimit_by_q_stop:nw
..... 28, 1548, 1549
\use_i_ii:nnn . 27, 400, 401, 1543,
1543, 1665, 2491, 17321, 17821, 17926
\use_ii:nn 26, 74, 393, 399,
531, 538, 863, 864, 960, 1232, 1235,
1249, 1253, 1254, 1266, 687, 692,
1432, 1499, 1500, 1592, 1698, 1733,
1738, 1925, 1927, 2093, 2426, 3830,
4406, 4762, 4893, 4914, 4932, 5109,
5456, 5578, 5755, 5838, 6156, 7420,
7443, 7517, 8730, 12516, 13224,
13301, 13307, 17628, 17630, 18639,
20798, 21532, 24565, 24588, 25009,
26378, 26643, 26644, 27278, 28611,
29097, 29103, 29107, 29615, 29848,
30281, 30386, 30641, 30745, 38658
\use_ii:nnn ... 26, 402, 968, 1501,
1502, 2373, 4404, 4760, 4890, 4911,
4929, 5063, 16835, 21047, 22625, 41923
\use_ii:nnnn
..... 26, 608, 609, 1501, 1505, 8499
\use_ii:nnnnn 26, 1501, 1509
\use_ii:nnnnnn 26, 1501, 1514
\use_ii:nnnnnnn 26, 1501, 1520
\use_ii:nnnnnnnn 26, 1501, 1527
\use_ii:nnnnnnnnn 26, 1501, 1535
\use_ii_i:nn 27, 786,
1544, 1544, 14576, 14661, 19649, 19739
\use_iii:nnn 26, 1501, 1503,
1936, 1938, 1944, 2393, 2431, 4799,
4940, 5252, 21144, 21151, 22626, 24368
\use_iii:nnnn . 26, 608, 609, 1501,
1506, 8499, 8521, 8523, 8524, 41816
\use_iii:nnnnn 26, 1501, 1510
\use_iii:nnnnnn 26, 1501, 1515
\use_iii:nnnnnnn 26, 1501, 1521
\use_iii:nnnnnnnn 26, 1501, 1528
\use_iii:nnnnnnnnn 26, 1501, 1536
\use_iv:nnnn 26,
608, 609, 1501, 1507, 8499, 8519, 26366
\use_iv:nnnnn 26, 1501, 1511
\use_iv:nnnnnn 26, 1501, 1516
\use_iv:nnnnnnn 26, 1501, 1522
\use_iv:nnnnnnnn 26, 1501, 1529
\use_iv:nnnnnnnnn 26, 1501, 1537
\use_ix:nnnnnnnnn 26, 1501, 1542
\use_none:n
..... 27, 382, 475, 477, 636, 674,
731, 805, 853, 907, 915, 918, 957,
959, 961, 963, 969, 971, 975, 976,
979, 982, 1100, 1101, 1104, 1498,
688, 694, 1552, 1552, 1664, 1716,
1717, 1736, 1737, 1856, 1863, 1900,
1907, 2120, 2310, 3007, 3008, 3826,
3875, 3990, 4028, 4179, 4200, 5623,
5918, 6047, 8723, 8728, 9085, 9465,
9659, 9876, 9880, 10131, 10806,
10862, 10918, 11196, 11254, 11613,
11616, 12550, 12601, 12608, 12722,
12812, 12903, 13156, 13167, 13228,
13263, 13267, 13292, 13471, 13480,
14512, 14535, 14551, 14563, 14596,
14731, 14763, 15017, 15027, 15065,

- 15127, 15292, 15376, 15481, 15653,
16830, 16983, 17079, 17094, 17218,
17234, 17318, 17377, 17830, 18062,
18063, 18641, 18796, 18802, 19148,
19151, 19221, 19266, 19369, 19458,
19485, 19524, 20484, 20827, 20849,
20909, 20916, 20918, 20922, 20924,
21099, 21157, 21167, 21174, 21179,
21185, 21305, 21314, 21318, 21321,
21325, 21367, 21462, 21558, 21569,
22716, 24144, 24159, 24357, 24507,
24511, 24515, 24519, 25825, 26078,
26085, 26102, 26121, 26144, 26212,
26253, 26378, 26393, 26414, 26415,
26705, 26706, 27252, 27255, 28259,
30002, 30290, 30803, 32366, 32682,
35725, 35738, 39303, 40228, 41737,
41740, 41884, 41885, 41904, 41943
`\use_none:nn` 27, 727,
733, 865, 870, 1567, 1552, 1553,
1646, 1654, 3116, 6446, 7155, 8477,
10863, 10907, 12707, 12893, 13058,
13258, 14620, 17608, 17637, 17852,
19190, 19499, 19646, 19735, 20988,
21406, 23388, 24422, 24506, 24510,
24514, 24518, 29997, 30652, 30831,
31066, 35726, 40721, 40739, 41881
`\use_none:nnn`
... 27, 742, 965, 973, 1552, 1554,
2888, 2903, 4059, 7633, 7910, 10864,
13215, 17207, 20977, 21226, 21228,
22857, 22866, 24505, 24509, 24513,
24517, 25166, 35730, 39518, 41761,
41771, 41798, 41807, 41826, 42419
`\use_none:nnnn` 27, 959, 961, 1552,
1555, 1870, 10865, 20767, 20854,
21849, 35728, 39824, 41745, 41753
`\use_none:nnnnn`
... 27, 397, 675, 676, 1084, 1552,
1556, 10866, 10876, 24638, 24674,
24704, 24712, 26831, 41831, 41837
`\use_none:nnnnnn`
.. 27, 1552, 1557, 1783, 10867, 13661
`\use_none:nnnnnnn`
27, 1084, 1552, 1558, 24640, 24676,
24706, 24714, 25050, 27292, 41890
`\use_none:nnnnnnnn`
..... 27, 401, 1552, 1559, 1687, 2969
`\use_none:nnnnnnnnn` .. 27, 1552, 1560
`\use_none_delimit_by_q_nil:w` ...
..... 27, 1545, 1545
`\use_none_delimit_by_q_recursion-
stop:w` . 27, 153, 399, 1545, 1547,
17071, 17086, 36128, 36162, 36171,
36196, 36220, 36233, 36255, 36270,
36300, 36312, 36332, 36344, 36360
`\use_none_delimit_by_q_stop:w` ...
..... 27, 805, 855, 1545, 1546
`\use_none_delimit_by_s_stop:w` ...
..... 155, 17367, 17367
`\use_v:nnnnn` 26, 1501, 1512
`\use_v:nnnnnn` 26, 1501, 1517
`\use_v:nnnnnnn` 26, 1501, 1523
`\use_v:nnnnnnnn` 26, 1501, 1530
`\use_v:nnnnnnnnn` 26, 1501, 1538
`\use_vi:nnnnnn` 26, 1501, 1518
`\use_vi:nnnnnnn` 26, 1501, 1524
`\use_vi:nnnnnnnn` 26, 1501, 1531
`\use_vi:nnnnnnnnn` 26, 1501, 1539
`\use_vii:nnnnnnn` 26, 1501, 1525
`\use_vii:nnnnnnnn` 26, 1501, 1532
`\use_vii:nnnnnnnnn` ... 26, 1501, 1540
`\use_viii:nnnnnnnnn` ... 26, 1501, 1533
`\use_viii:nnnnnnnnnn` .. 26, 1501, 1541
`\useboxresource` 955
`\usefont` 35728
`\useimageresource` 956
`\Uskewed` 1123
`\Uskewedwithdelims` 1124
`\Ustack` 1125
`\Ustartdisplaymath` 1126
`\Ustartmath` 1127
`\Ustopdisplaymath` 1128
`\Ustopmath` 1129
`\Usubscript` 1130
`\Usuperscript` 1131
`\Uunderdelimater` 1132
`\Uvextensible` 1133
- ## V
- `\v` 33333, 34932, 35799, 35820,
35906, 35907, 35908, 35909, 35918,
35919, 35952, 35953, 35960, 35961,
35972, 35973, 35980, 35981, 35984,
35985, 36007, 36008, 36009, 36010,
36011, 36012, 36013, 36014, 36015,
36016, 36017, 36018, 36019, 36020,
36021, 36024, 36025, 36034, 36035
`\vadjust` 449
`\valign` 450
value commands:
 `.value_forbidden:n` 251, 23128
 `.value_required:n` 251, 23128
`\variablefam` 927
`\vbadness` 451
`\vbox` 1499, 452

vbox commands:

`\vbox:n` 312,
 316, 36616, 36616, 37189, 37196, 37231
`\vbox:nn` 317, 36695, 36709
`\vbox:nw` 317, 36695, 36695
`\vbox:w` 316, 36616, 36620
`\vbox_center:n` 316, 36632, 36650
`\vbox_center:nn` ... 317, 36632, 36648
`\vbox_center:nw` ... 317, 36632, 36632
`\vbox_center:w` 316, 36632, 36634
`\vbox_center_to_ht:nn`
 317, 36632, 36652
`\vbox_center_to_ht:nw`
 317, 36632, 36636
`\vbox_end:`
 316, 317, 36616, 36624, 36776, 36777
`\vbox_gset:Nn`
 317, 36726, 36731, 36737, 37471, 42291
`\vbox_gset:Nw`
 318, 36762, 36768, 36775, 37546, 42295
`\vbox_gset_end:`
 318, 36762, 36777, 37548
`\vbox_gset_split_to_ht:NNn`
 318, 36796, 36799, 36804, 42297
`\vbox_gset_to_ht:Nnn`
 318, 36750, 36755, 36761, 42293
`\vbox_gset_to_ht:Nnw`
 318, 36778, 36784, 36791, 42296
`\vbox_gset_top:Nn`
 318, 36738, 36743, 36749, 42292
`\vbox_gset_top_to_ht:Nnn` 42294
`\vbox_set:Nn`
 .. 317, 318, 36649, 36651, 36710,
 36726, 36726, 36736, 37465, 42209
`\vbox_set:Nw`
 318, 36762, 36762, 36774, 37539, 42213
`\vbox_set_end:` 318, 36762, 36776, 37541
`\vbox_set_split_to_ht:NNn`
 318, 36796, 36796, 36798, 42215
`\vbox_set_to_ht:Nnn`
 318, 36653, 36750, 36750, 36760, 42211
`\vbox_set_to_ht:Nnw`
 318, 36778, 36778, 36790, 42214
`\vbox_set_top:Nn` 318, 36712, 36738,
 36738, 36748, 37485, 37562, 42210
`\vbox_set_top_to_ht:Nnn` 42212
`\vbox_to_ht:nn`
 317, 36674, 36674, 36694, 40658
`\vbox_to_ht:nw` 317, 36674, 36683
`\vbox_to_zero:n` ... 317, 36674, 36693
`\vbox_top:n` 316, 36616, 36618
`\vbox_top:nn` 317, 36695, 36711
`\vbox_top:nw` 317, 36695, 36697
`\vbox_top:w` 316, 36616, 36622

`\vbox_top_to_ht:nn` . 317, 36674, 36676
`\vbox_top_to_ht:nw` . 317, 36674, 36685
`\vbox_unpack:N`
 318, 36792, 36792, 36794, 37485, 37562
`\vbox_unpack_drop:N`
 319, 36792, 36793, 36795
`\vcenter` 453
vcoffin commands:
`\vcoffin_gset:Nnn`
 326, 37462, 37468, 37473
`\vcoffin_gset:Nnw`
 326, 37537, 37544, 37550
`\vcoffin_gset_end:`
 326, 37537, 37547, 37578
`\vcoffin_set:Nnn`
 326, 37462, 37462, 37467
`\vcoffin_set:Nnw`
 326, 37537, 37537, 37543
`\vcoffin_set_end:`
 326, 37537, 37540, 37577
`\vfi` 1199
`\vfil` 454
`\vfill` 455
`\vfilneg` 456
`\vfuzz` 457
`\voffset` 458
`\vpack` 928
`\vrule` 459
`\vsize` 460
`\vskip` 461
`\vsplit` 462
`\vss` 463
`\vtop` 464

W

`\wd` 465
`\widowpenalties` 537
`\widowpenalty` 466
`\wordboundary` 929
`\write` 68, 467

X

`\xdef` 468
`\XeTeXcharclass` 705
`\XeTeXcharglyph` 706
`\XeTeXcountfeatures` 707
`\XeTeXcountglyphs` 708
`\XeTeXcountselectors` 709
`\XeTeXcountvariations` 710
`\XeTeXdashbreakstate` 712
`\XeTeXdefaultencoding` 711
`\XeTeXfeaturecode` 713
`\XeTeXfeaturename` 714
`\XeTeXfindfeaturebyname` 715

\XeTeXfindselectorbyname	717	\XeTeXpdffile	749
\XeTeXfindvariationbyname	719	\XeTeXpdfpagecount	750
\XeTeXfirstfontchar	721	\XeTeXpicfile	751
\XeTeXfonttype	722	\XeTeXprotrudechars	778
\XeTeXgenerateactualtext	723	\XeTeXrevision	752
\XeTeXglyph	725	\XeTeXselectorcode	763
\XeTeXglyphbounds	726	\XeTeXselectorname	753
\XeTeXglyphindex	727	\XeTeXtracingfonts	754
\XeTeXglyphname	728	\XeTeXupwardsmode	755
\XeTeXhyphenatablelength	766	\XeTeXuseglyphmetrics	756
\XeTeXinputencoding	729	\XeTeXvariation	757
\XeTeXinputnormalization	730	\XeTeXvariationdefault	758
\XeTeXinterchartokenstate	732	\XeTeXvariationmax	759
\XeTeXinterchartoks	734	\XeTeXvariationmin	760
\XeTeXinterwordspaceshaping	764	\XeTeXvariationname	761
\XeTeXisdefaultselector	735	\XeTeXversion	762
\XeTeXisexclusivefeature	737	\xkanjiskip	1195
\XeTeXlastfontchar	739	\xleaders	469
\XeTeXlinebreaklocale	741	\xspaceskip	470
\XeTeXlinebreakpenalty	742	\xspacecode	1196
\XeTeXlinebreakskip	740	\xtoksapp	930
\XeTeXOTcountfeatures	743	\xtokspre	931
\XeTeXOTcountlanguages	744		
\XeTeXOTcountscripts	745		
\XeTeXOTfeaturetag	746		
\XeTeXOTlanguagegetag	747		
\XeTeXOTscripttag	748		
		Y	
		\ybaselineshift	1197
		\year	471, 1299, 9100
		\yoko	1198