

The `expl3` package and \LaTeX 3 programming

The \LaTeX Project*

Released 2026-03-16

Abstract

This document gives an introduction to a new set of programming conventions that have been designed to meet the requirements of implementing large scale \TeX macro programming projects such as \LaTeX . These programming conventions are the base layer of \LaTeX 3.

The main features of the system described are:

- classification of the macros (or, in \LaTeX terminology, commands) into \LaTeX functions and \LaTeX parameters, and also into modules containing related commands;
- a systematic naming scheme based on these classifications;
- a simple mechanism for controlling the expansion of a function's arguments.

This system is being used as the basis for \TeX programming within The \LaTeX Project. Note that the language is not intended for either document mark-up or style specification. Instead, it is intended that such features will be built on top of the conventions described here.

This document is an introduction to the ideas behind the `expl3` programming interface. For the complete documentation of the programming layer provided by The \LaTeX Project, see the accompanying `interface3` document.

1 Introduction

The first step to develop a \LaTeX kernel beyond \LaTeX 2 ϵ is to address how the underlying system is programmed. Rather than the current mix of \LaTeX and \TeX macros, the \LaTeX 3 system provides its own consistent interface to all of the functions needed to control \TeX . A key part of this work is to ensure that everything is documented, so that \LaTeX programmers and users can work efficiently without needing to be familiar with the internal nature of the kernel or with plain \TeX .

The `expl3` bundle provides this new programming interface for \LaTeX . To make programming systematic, \LaTeX 3 uses some very different conventions to \LaTeX 2 ϵ or plain \TeX . As a result, programmers starting with \LaTeX 3 need to become familiar with the syntax of the new language.

The next section shows where this language fits into a complete \TeX -based document processing system. We then describe the major features of the syntactic structure of command names, including the argument specification syntax used in function names.

The practical ideas behind this argument syntax will be explained, together with the expansion control mechanism and the interface used to define variant forms of functions.

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

As we shall demonstrate, the use of a structured naming scheme and of variant forms for functions greatly improves the readability of the code and hence also its reliability. Moreover, experience has shown that the longer command names which result from the new syntax do not make the process of *writing* code significantly harder.

2 Languages and interfaces

It is possible to identify several distinct languages related to the various interfaces that are needed in a T_EX-based document processing system. This section looks at those we consider most important for the L^AT_EX3 system.

Document mark-up This comprises those commands (often called tags) that are to be embedded in the document (the `.tex` file).

It is generally accepted that such mark-up should be essentially *declarative*. It may be traditional T_EX-based mark-up such as L^AT_EX 2_ε, as described in [3] and [2], or a mark-up language defined via HTML or XML.

One problem with more traditional T_EX coding conventions (as described in [1]) is that the names and syntax of T_EX's primitive formatting commands are ingeniously designed to be “natural” when used directly by the author as document mark-up or in macros. Ironically, the ubiquity (and widely recognized superiority) of logical mark-up has meant that such explicit formatting commands are almost never needed in documents or in author-defined macros. Thus they are used almost exclusively by T_EX programmers to define higher-level commands, and their idiosyncratic syntax is not at all popular with this community. Moreover, many of them have names that could be very useful as document mark-up tags were they not pre-empted as primitives (*e.g.* `\box` or `\special`).

Designer interface This relates a (human) typographic designer's specification for a document to a program that “formats the document”. It should ideally use a declarative language that facilitates expression of the relationship and spacing rules specified for the layout of the various document elements.

This language is not embedded in document text and it will be very different in form to the document mark-up language. For L^AT_EX, this level was almost completely missing from L^AT_EX2.09; L^AT_EX 2_ε made some improvements in this area but it is still the case that implementing a design specification in L^AT_EX requires far more “low-level” coding than is acceptable.

Programmer interface This language is the implementation language within which the basic typesetting functionality is implemented, building upon the primitives of T_EX (or a successor program). It may also be used to implement the previous two languages “within” T_EX, as in the current L^AT_EX system.

The last layer is covered by the conventions described in this document, which describes a system aimed at providing a suitable basis for coding L^AT_EX3. Its main distinguishing features are summarized here:

- A consistent naming scheme for all commands, including T_EX primitives.
- The classification of commands as L^AT_EX functions or L^AT_EX parameters, and also their division into modules according to their functionality.

- A simple mechanism for controlling argument expansion.
- Provision of a set of core L^AT_EX functions that is sufficient for handling programming constructs such as queues, sets, stacks, property lists.
- A T_EX programming environment in which, for example, all white space is ignored.

3 The naming scheme

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. In contrast to the plain T_EX format and the L^AT_EX 2_ε kernel, these extra letters are used only between parts of a macro name (no strange vowel replacement).

While T_EX is actually a macro processor, by convention for the expl3 programming language we distinguish between *functions* and *variables*. Functions can have arguments and they are either expanded or executed. Variables can be assigned values and they are used in arguments to functions; they are not used directly but are manipulated by functions (including getting and setting functions). Functions and variables with a related functionality (for example accessing counters, or manipulating token lists, etc.) are collected together into a *module*.

3.1 Examples

Before giving the details of the naming scheme, here are a few typical examples to indicate the flavor of the scheme; first some variable names.

`\l_tmpa_box` is a local variable (hence the `l_` prefix) corresponding to a box register.
`\g_tmpa_int` is a global variable (hence the `g_` prefix) corresponding to an integer register (i.e., a T_EX count register).
`\c_empty_tl` is the constant (`c_`) token list variable that is always empty.

Now here is an example of a typical function name.

`\seq_push:Nn` is the function which puts the token list specified by its second argument onto the stack specified by its first argument. The different natures of the two arguments are indicated by the `:Nn` suffix. The first argument must be a single token which “names” the stack parameter: such single-token arguments are denoted `N`. The second argument is a normal T_EX “undelimited argument”, which may either be a single token or a balanced, brace-delimited token list (which we shall here call a *braced token list*): the `n` denotes such a “normal” argument form. The name of the function indicates it belongs to the `seq` module.

3.2 Formal naming syntax

We shall now look in more detail at the syntax of these names. A function name in L^AT_EX3 has a name consisting of three parts:

`\<module>_<description>:<arg-spec>`

while a variable has (up to) four distinct parts to its name:

`\<scope>_<module>_<description>_<type>`

The syntax of all names contains

`<module>` and `<description>`

and these two both give information about the command.

A *module* is a collection of closely related functions and variables. Typical module names include `int` for integer parameters and related functions, `seq` for sequences and `box` for boxes.

Packages providing new programming functionality will add new modules as needed; the programmer can choose any unused name, consisting of letters only, for a module. In general, the module name and module prefix should be related: for example, the kernel module containing `box` functions is called `l3box`. Module names and programmers' contact details are listed in `l3prefixes.csv`.

The *description* gives more detailed information about the function or parameter, and provides a unique name for it. It should consist of letters and, possibly, `_` characters. In general, the description should use `_` to divide up “words” or other easy to follow parts of the name. For example, the L^AT_EX3 kernel provides `\if_cs_exist:N` which, as might be expected, tests if a command name exists.

Where functions for variable manipulation can perform assignments either locally or globally, the latter case is indicated by the inclusion of a `g` in the second part of the function name. Thus `\tl_set:Nn` is a local function but `\tl_gset:Nn` acts globally. Functions of this type are always documented together, and the scope of action may therefore be inferred from the presence or absence of a `g`. See the next subsection for more detail on variable scope.

3.2.1 Separating private and public material

One of the issues with the T_EX language is that it doesn't support name spaces and encapsulation other than by convention. As a result nearly every internal command in the L^AT_EX 2_ε kernel has eventually be used by extension packages as an entry point for modifications or extensions. The consequences of this is that nowadays it is next to impossible to change anything in the L^AT_EX 2_ε kernel (even if it is clearly just an internal command) without breaking something.

In `expl3` we hope to improve this situation drastically by clearly separating public interfaces (that extension packages can use and rely on) and private functions and variables (that should not appear outside of their module). There is (nearly) no way to enforce this without severe computing overhead, so we implement it only through a naming convention, and some support mechanisms. However, we think that this naming convention is easy to understand and to follow, so that we are confident that this will adopted and provides the desired results.

Functions created by a module may either be “public” (documented with a defined interface) or “private” (to be used only within that module, and thus not formally documented). It is important that only documented interfaces are used; at the same time, it is necessary to show within the name of a function or variable whether it is public or private.

To allow clear separation of these two cases, the following convention is used. To denote a private function or a private variable (of the module), two `_` characters are used in front of the module name, e.g.

```
\module_foo:nnn
```

is a public function which should be documented while

```
\__module_foo:nnn
```

is private to the module, and should *not* be used outside of that module.

For variables, to avoid three `_` in a row, the separator for the variable scope and any leading `_` for a private interface in the module part are combined. Thus

```
\l_module_foo_tl
```

is a public variable and

```
\l__module_foo_tl
```

is private.

3.2.2 Using `@@` and `DocStrip` to mark private code

The formal syntax for internal functions allows clear separation of public and private code, but includes redundant information (every internal function or variable includes `__<module>`). To aid programmers, the `DocStrip` program introduces the syntax

```
%<@@=<module>
```

which then allows `@@` (and `__@@` in case of variables) to be used as a place holder for `__<module>` in code. Thus for example

```
%<@@=foo>
%   \begin{macrocode}
\cs_new:Npn \@@_function:n #1
...
\tl_new:N \l_@@_my_tl
%   \end{macrocode}
```

is converted by `DocStrip` to

```
\cs_new:Npn \__foo_function:n #1
...
\tl_new:N \l__foo_my_tl
```

on extraction. As you can see both `__@@` and `@@` are mapped to `__<module>`, because we think that this helps to distinguish variables from functions in the source when the `@@` convention is used.

3.2.3 Variables: declaration

In well-formed `expl3` code, variables should always be declared before assignment is attempted. This is true even for variable types where the underlying `TeX` implementation will allow direct assignment. This applies both to setting directly (`\tl_set:Nn`, etc.) and to setting equal (`\tl_set_eq:NN`, etc.).

To help programmers to adhere to this approach, the debugging option `check-declarations` may be given

```
\debug_on:n { check-declarations }
```

and will issue an error whenever an assignment is made to a non-declared variable. There is a performance implication, so this option should only be used for testing.

3.2.4 Variables: scope and type

The $\langle scope \rangle$ part of the name describes how the variable can be accessed. Variables are classified as local, global or constant. This *scope* type appears as a code at the beginning of the name; the codes used are:

- c** constants (global variables whose value should not be changed);
- g** variables whose value should only be set globally;
- l** variables whose value should only be set locally.

Separate functions are provided to assign data to local and global variables; for example, `\tl_set:Nn` and `\tl_gset:Nn` respectively set the value of a local or global “token list” variable. Note that it is a poor T_EX practice to intermix local and global assignments to a variable; otherwise you risk exhausting the save stack.¹

The $\langle type \rangle$ is in the list of available *data-types*;² these include the primitive T_EX data-types, such as the various registers, but to these are added data-types built within the L^AT_EX programming system.

The data types in L^AT_EX3 are:

- bitset** a string of bits (0 and 1 tokens) that are accessed by position or by name;
- bool** either true or false (the L^AT_EX3 implementation does not use `\iftrue` or `\iffalse`);
- box** box register;
- cctab** category code table;
- clist** comma separated list;
- coffin** a “box with handles” — a higher-level data type for carrying out **box** alignment operations;
- dim** “rigid” lengths;
- fp** floating-point values;
- fparray** fixed-size vector of floating-point values;
- int** integer-valued count register;
- intarray** fixed-size vector of integer values;
- ior** an input stream (for reading from a file);
- iow** an output stream (for writing to a file);
- muskip** math mode “rubber” lengths;
- prop** property list;
- regex** regular expression;
- seq** sequence: a data-type used to implement lists (with access at both ends) and stacks;

¹See *The T_EXbook*, p. 301, for further information.

²Of course, if a totally new data type is needed then this will not be the case. However, it is hoped that only the kernel team will need to create new data types.

skip “rubber” lengths;

str T_EX strings: a special case of **t1** in which all characters have category “other” (catcode 12), other than spaces which are category “space” (catcode 10);

token equal to a single arbitrary token;

t1 “token list variables”: placeholders for token lists.

When the *<type>* and *<module>* are identical (as often happens in the more basic modules) the *<module>* part is often omitted for aesthetic reasons.

The name “token list” may cause confusion, and so some background is useful. T_EX works with tokens and lists of tokens, rather than characters. It provides two ways to store these token lists: within macros and as token registers (**toks**). The implementation in L^AT_EX3 means that **toks** are not required, and that all operations for storing tokens can use the **t1** variable type.

There are two more special types of constants:

q constants of quark type;

s constants of scan mark type.

The *<type>* part of quark and scan mark constants is omitted, and the *<module>* part of general quarks and scan marks is often omitted too.

Experienced T_EX programmers will notice that some of the variable types listed are native T_EX registers whilst others are not. In general, the underlying T_EX implementation for a data structure may vary but the *documented interface* will be stable. For example, the **prop** data type was originally implemented as a **toks**, but is currently built on top of the **t1** data structure.

3.2.5 Variables: guidance

Both comma lists and sequences have similar characteristics. They both use special delimiters to mark out one entry from the next, and are both accessible at both ends. In general, it is easier to create comma lists ‘by hand’ as they can be typed in directly. User input often takes the form of a comma separated list and so there are many cases where this is the obvious data type to use. On the other hand, sequences use special internal tokens to separate entries. This means that they can be used to contain material that comma lists cannot (such as items that may themselves contain commas!). In general, comma lists should be preferred for creating fixed lists inside programs and for handling user input where commas will not occur. On the other hand, sequences should be used to store arbitrary lists of data.

expl3 implements stacks using the sequence data structure. Thus creating stacks involves first creating a sequence, and then using the sequence functions which work in a stack manner (**\seq_push:Nn**, *etc.*).

Due to the nature of the underlying T_EX implementation, it is possible to assign values to token list variables and comma lists without first declaring them. However, this is *not supported behavior*. The L^AT_EX3 coding convention is that all variables must be declared before use.

The **expl3** package can be loaded with the **check-declarations** option to verify that all variables are declared before use. This has a performance implication and is therefore intended for testing during development and not for use in production documents.

3.2.6 Functions: argument specifications

Function names end with an `<arg-spec>` after a colon. This gives an indication of the types of argument that a function takes, and provides a convenient method of naming similar functions that differ only in their argument forms (see the next section for examples).

The `<arg-spec>` consists of a (possibly empty) list of letters, each denoting one argument of the function. The letter, including its case, conveys information about the type of argument required.

All functions have a base form with arguments using one of the following argument specifiers:

- n** Unexpanded token or braced token list.
This is a standard \TeX undelimited macro argument.
- N** Single token (unlike **n**, the argument must *not* be surrounded by braces).
A typical example of a command taking an **N** argument is `\cs_set`, in which the command being defined must be unbraced.
- p** Primitive \TeX parameter specification.
This can be something simple like `#1#2#3`, but may use arbitrary delimited argument syntax such as: `#1,#2\q_stop#3`. This is used when defining functions.
- T,F** These are special cases of **n** arguments, used for the true and false code in conditional commands.

There are two other specifiers with more general meanings:

- D** Stands for **Do not use**. This special case is used for \TeX primitives. 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](#)³.
- w** This means that the argument syntax is “weird” in that it does not follow any standard rule. It is used for functions with arguments that take non standard forms: examples are \TeX -level delimited arguments and the boolean tests needed after certain primitive `\if...` commands.

In case of **n** arguments that consist of a single token the surrounding braces can be omitted in nearly all situations—functions that force the use of braces even for single token arguments are explicitly mentioned. However, programmers are encouraged to always use braces around **n** arguments, as this makes the relationship between function and argument clearer.

Further argument specifiers are available as part of the expansion control system. These are discussed in the next section.

³If a primitive offers a functionality not yet in the kernel, programmers and users are encouraged to write to the team describing their use-case and intended behavior, so that a possible interface can be discussed. Temporarily, while an interface is not provided, programmers may use the procedure described in the [l3styleguide.pdf](#).

4 Expansion control

Let's take a look at some typical operations one might want to perform. Suppose we maintain a stack of open files and we use the stack `\g_ior_file_name_seq` to keep track of them (`ior` is the prefix used for the file reading module). The basic operation here is to push a name onto this stack which could be done by the operation

```
\seq_gpush:Nn \g_ior_file_name_seq {#1}
```

where `#1` is the filename. In other words, this operation would push the file name as is onto the stack.

However, we might face a situation where the filename is stored in a variable of some sort, say `\l_ior_curr_file_tl`. In this case we want to retrieve the value of the variable. If we simply use

```
\seq_gpush:Nn \g_ior_file_name_seq \l_ior_curr_file_tl
```

we do not get the value of the variable pushed onto the stack, only the variable name itself. Instead a suitable number of `\exp_after:wN` would be necessary (together with extra braces) to change the order of expansion,⁴ i.e.

```
\exp_after:wN
  \seq_gpush:Nn
\exp_after:wN
  \g_ior_file_name_seq
\exp_after:wN
  { \l_ior_curr_file_tl }
```

The above example is probably the simplest case but already shows how the code changes to something difficult to understand. Furthermore there is an assumption in this: that the storage bin reveals its contents after exactly one expansion. Relying on this means that you cannot do proper checking plus you have to know exactly how a storage bin acts in order to get the correct number of expansions. Therefore $\text{\LaTeX}3$ provides the programmer with a general scheme that keeps the code compact and easy to understand.

To denote that some argument to a function needs special treatment one just uses different letters in the arg-spec part of the function to mark the desired behavior. In the above example one would write

```
\seq_gpush:NV \g_ior_file_name_seq \l_ior_curr_file_tl
```

to achieve the desired effect. Here the `V` (the second argument) is for “retrieve the value of the variable” before passing it to the base function.

The following letters can be used to denote special treatment of arguments before passing it to the base function:

c Character string used as a command name.

The argument (a token or braced token list) is *fully expanded*; the result must be a sequence of characters which is then used to construct a command name (*via* `\csname ... \endcsname`). This command name is a single token that is passed to the function as the argument. Hence

⁴`\exp_after:wN` is the $\text{\LaTeX}3$ name for the \TeX `\expandafter` primitive.

```
\seq_gpush:cV { g_file_name_seq } \l_tmpa_tl
```

is equivalent to

```
\seq_gpush:NV \g_file_name_seq \l_tmpa_tl.
```

Full expansion means that (a) the entire argument must be expandable and (b) any variables are converted to their content. So the preceding examples are also equivalent to

```
\tl_new:N \g_file_seq_name_tl  
\tl_gset:Nn \g_file_seq_name_tl { g_file_name_seq }  
\seq_gpush:cV { \tl_use:N \g_file_seq_name_tl } \l_tmpa_tl.
```

(Token list variables are expandable and we could omit the accessor function `\tl_use:N`. Other variable types require the appropriate `\<var>_use:N` functions to be used in this context.)

V Value of a variable.

This means that the contents of the register in question is used as the argument, be it an integer, a length-type register, a token list variable or similar. The value is passed to the function as a braced token list. Can be applied to variables which have a `\<var>_use:N` function (other than boxes), and which therefore deliver a single “value”.

v Value of a register, constructed from a character string used as a command name.

This is a combination of **c** and **V** which first constructs a control sequence from the argument and then passes the value of the resulting register to the function. Can be applied to variables which have a `\<var>_use:N` function (other than boxes), and which therefore deliver a single “value”.

e Fully-expanded token or braced token list.

This means that the argument is expanded as in the replacement text of a `\message`, and the expansion is passed to the function as a braced token list.

o One-level-expanded token or braced token list.

This means that the argument is expanded one level, as by `\expandafter`, and the expansion is passed to the function as a braced token list. Note that if the original argument is a braced token list then only the first token in that list is expanded. In general, using **V** should be preferred to using **o** for simple variable retrieval.

f Expanding the first token recursively in a braced token list.

Almost the same as the **e** type except here the token list is expanded fully until the first unexpandable token is found and the rest is left unchanged. Note that if this function finds a space at the beginning of the argument it gobbles it and does not expand the next token.

x Fully-expanded token or braced token list.

This expansion is very similar to **e**-type but is not nestable, can only be used to create non-expandable functions, and requires that `#` tokens are doubled in the argument. In almost all cases, **e**-type should be preferred: retained largely for historical reasons, and should where possible be replaced by the **e**-type equivalent.

4.1 Simpler means better

Anyone who programs in T_EX is frustratingly familiar with the problem of arranging that arguments to functions are suitably expanded before the function is called. To illustrate how expansion control can bring instant relief to this problem we shall consider two examples copied from `latex.ltx`.

```
\global\expandafter\let
\csname\cf@encoding\string#1\expandafter\endcsname
\csname ?\string#1\endcsname
```

This first piece of code is in essence simply a global `\let` whose two arguments firstly have to be constructed before `\let` is executed. The `#1` is a control sequence name such as `\textcurrency`. The token to be defined is obtained by concatenating the characters of the current font encoding stored in `\cf@encoding`, which has to be fully expanded, and the name of the symbol. The second token is the same except it uses the default encoding `?`. The result is a mess of interwoven `\expandafter` and `\csname` beloved of all T_EX programmers, and the code is essentially unreadable.

Using the conventions and functionality outlined here, the task would be achieved with code such as this:

```
\cs_gset_eq:cc
{ \cf@encoding \token_to_str:N #1 } { ? \token_to_str:N #1 }
```

The command `\cs_gset_eq:cc` is a global `\let` that generates command names out of both of its arguments before making the definition. This produces code that is far more readable and more likely to be correct first time. (`\token_to_str:N` is the L^AT_EX3 name for `\string`.)

Here is the second example.

```
\expandafter
\in@
\csname sym#3%
\expandafter
\endcsname
\expandafter
{ %
\group@list } %
```

This piece of code is part of the definition of another function. It first produces two things: a token list, by expanding `\group@list` once; and a token whose name comes from `'sym#3'`. Then the function `\in@` is called and this tests if its first argument occurs in the token list of its second argument.

Again we can improve enormously on the code. First we shall rename the function `\in@`, which tests if its first argument appears within its second argument, according to our conventions. Such a function takes two normal “n” arguments and operates on token lists: it might reasonably be named `\tl_test_in:nn`. Thus the variant function we need would be defined with the appropriate argument types and its name would be `\tl_test_in:cV`. Now this code fragment would be simply:

```
\tl_test_in:cV { sym #3 } \group@list
```

This code could be improved further by using a sequence `\l_group_seq` rather than the bare token list `\group@list`. Note that, in addition to the lack of `\expandafter`, the space after the `}` is silently ignored since all white space is ignored in this programming environment.

4.2 New functions from old

For many common functions the L^AT_EX3 kernel provides variants with a range of argument forms, and similarly it is expected that extension packages providing new functions will make them available in all the commonly needed forms.

However, there will be occasions where it is necessary to construct a new such variant form; therefore the expansion module provides a straightforward mechanism for the creation of functions with any required argument type, starting from a function that takes “normal” T_EX undelimited arguments.

To illustrate this let us suppose you have a “base function” `\demo_cmd:Nnn` that takes three normal arguments, and that you need to construct the variant `\demo_cmd:cne`, for which the first argument is used to construct the *name* of a command, whilst the third argument must be fully expanded before being passed to `\demo_cmd:Nnn`. To produce the variant form from the base form, simply use this:

```
\cs_generate_variant:Nn \demo_cmd:Nnn { cne }
```

This defines the variant form so that you can then write, for example:

```
\demo_cmd:cne { abc } { pq } { \rst \xyz }
```

rather than ... well, something like this!

```
\def \tempa {{pq}}%
\edef \tempb {\rst \xyz}%
\expandafter
  \demo@cmd:nnn
\csname abc%
  \expandafter
  \expandafter
  \expandafter
  \endcsname
\expandafter
  \tempa
\expandafter
  {%
\tempb
  }%
```

Another example: you may wish to declare a function `\demo_cmd_b:enene`, a variant of an existing function `\demo_cmd_b:nnnnn`, that fully expands arguments 1, 3 and 5, and produces commands to pass as arguments 2 and 4 using `\csname`. The definition you need is simply

```
\cs_generate_variant:Nn \demo_cmd_b:nnnnn { enene }
```

This extension mechanism is written so that if the same new form of some existing command is implemented by two extension packages then the two definitions are identical and thus no conflict occurs.

5 The distribution

The `expl3` modules are designed to be loaded on top of $\text{\LaTeX} 2_{\epsilon}$.

The core `expl3` language is broadly stable, and thus the syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`. A robust mechanism is in place for such deprecations.

The distribution of `expl3` is split up into three packages on CTAN: `l3kernel`, `l3packages` and `l3experimental`. The core programming layer provided by `l3kernel` has been loaded as part of the \LaTeX since 2020-02-02. For historical reasons, in older kernel releases

```
\RequirePackage{expl3}
```

loads the code distributed as `l3kernel`. This monolithic package contains all of the modules regarded by the team as stable, and any changes in this code are very limited. This material is therefore suitable for use in third-party packages without concern about changes in support. All of this code is documented in `interface3.pdf`.

The material in `l3packages` is also stable; this bundle provides user-level commands, some of which have been integrated in the \LaTeX kernel.

Finally, `l3experimental` contains modules ready for public use but not yet integrated into `l3kernel`. These modules have to be loaded explicitly. The team anticipate that all of these modules will move to stable status over time, but they may be more flexible in terms of interface and functionality detail. Feedback on these modules is extremely valuable.

6 Moving from $\text{\LaTeX} 2_{\epsilon}$ to `expl3`

To help programmers to use `expl3` code in existing $\text{\LaTeX} 2_{\epsilon}$ package, some short notes on making the change are probably desirable. Suggestions for inclusion here are welcome! Some of the following is concerned with code, and some with coding style.

- `expl3` is mainly focused on programming. This means that some areas still require the use of $\text{\LaTeX} 2_{\epsilon}$ internal macros. For example, you may well need `\IfPackageLoadedTF`, as there is currently no native `expl3` package loading module.
- User level macros should be generated using the mechanism available in the `ltxcmd` module, which is part of the the \LaTeX kernel since 2020-10-01.
- At an internal level, most functions should be generated `\long` (using `\cs_new:Npn`) rather than “short” (using `\cs_new_nopar:Npn`).
- Where possible, declare all variables and functions (using `\cs_new:Npn`, `\tl_new:N`, etc.) before use.
- Prefer “higher-level” functions over “lower-level”, where possible. So for example use `\cs_if_exist:NTF` and not `\if_cs_exist:N`.
- Use space to make code readable. In general, we recommend a layout such as:

```

\cs_new:Npn \foo_bar:Nn #1#2
{
  \cs_if_exist:NTF #1
  { \__foo_bar:n {#2} }
  { \__foo_bar:nm {#2} { literal } }
}

```

where spaces are used around { and } except for isolated #1, #2, etc.

- Put different code items on separate lines: readability is much more useful than compactness.
- Use long, descriptive names for functions and variables, and for auxiliary functions use the parent function name plus `aux`, `auxi`, `auxii` and so on.
- If in doubt, ask the team: someone will soon get back to you!

7 Load-time options for `expl3`

To support code authors, the `expl3` package for $\text{\LaTeX} 2_{\epsilon}$ includes a small number of load-time options. These all work in a key–value sense, recognizing the `true` and `false` values. Giving the option name alone is equivalent to using the option with the `true` value.

check-declarations All variables used in `expl3` code should be declared. This is enforced by \TeX for variable types based on \TeX registers, but not for those which are constructed using macros as the underlying storage system. The `check-declarations` option enables checking for all variable assignments, issuing an error if any variables are assigned without being initialized. See also `\debug_on:n {check-declarations}` in `interface3` for finer control.

log-functions The `log-functions` option is used to enable recording of every new function name in the `.log` file. This is useful for debugging purposes, as it means that there is a complete list of all functions created by each module loaded (with the exceptions of a very small number required by the bootstrap code). See also `\debug_on:n {log-functions}` in `interface3` for finer control.

backend Selects the backend to be used for color, graphics and related operations that are backend-dependent. Options available are

`dvips` Use the `dvips` driver.

`dvipdfmx` Use the `dvipdfmx` driver.

`dvisvgm` Use the `dvisvgm` driver.

`luatex` Use the direct PDF output mode of \LuaTeX

`pdftex` Use the direct PDF output mode of \pdfTeX

`xetex` Use the \XeTeX version of the `dvipdfmx` driver.

For historical reasons, there is also `pdfmode` as an equivalent of `luatex` or `pdftex`, and `xdvipdfmx` as an equivalent to `xetex`, but these are deprecated

suppress-backend-headers The `suppress-backend-headers` option suppresses loading of backend-specific header files; currently this only affects `dvips`. This option is available to support DVI-based routes that do not support the `header` line used by `dvips`.

The debugging options may also be given using `\keys_set:nn { sys } { ... }`; the `backend` option can be given in this way *only* if a backend has not already been loaded. This method of setting options is useful where `expl3` is pre-loaded by the $\text{\LaTeX} 2_{\epsilon}$ format.

8 Using `expl3` with formats other than $\text{\LaTeX} 2_{\epsilon}$

As well as the $\text{\LaTeX} 2_{\epsilon}$ package `expl3`, there is also a “generic” loader for the code, `expl3-generic.tex`. This may be loaded using the plain \TeX syntax

```
\input expl3-generic %
```

This enables the programming layer to work with the other formats. As no options are available loading in this way, the “native” drivers are automatically used. If this “generic” loader is used with $\text{\LaTeX} 2_{\epsilon}$ the code automatically switches to the appropriate package route.

After loading the programming layer using the generic interface, the commands `\ExplSyntaxOn` and `\ExplSyntaxOff` and the code-level functions and variables detailed in `interface3` are available. Note that other $\text{\LaTeX} 2_{\epsilon}$ packages *using* `expl3` are not loadable: package loading is dependent on the $\text{\LaTeX} 2_{\epsilon}$ package-management mechanism.

9 Getting the version of `expl3`

`\ExplLoaderFileDate` Once the programming layer is loaded by one of the loaders, you can access its version in the ISO date format `\year-\month-\day`, through `\ExplLoaderFileDate`.

The current version of `expl3` is 2026-03-16.

10 Engine/primitive requirements

To use `expl3` and the higher level packages provided by the team, the minimal set of primitive requirements is currently described in [README.md](#).

Practically, these requirements are met by the engines

- $\text{pdf}\text{\TeX}$ v1.40.20 or later.
- $\text{X}\text{\TeX}$ v0.999991 or later.
- $\text{Lua}\text{\TeX}$ v1.10 or later.
- $\text{e-(u)}\text{p}\text{\TeX}$ v3.8.2 or later.
- Prote (2021) or later.

Additional modules beyond the core of `expl3` may require additional primitives. In particular, third-party authors may significantly extend the primitive coverage requirements.

<code>\iffalse</code>	6	tl commands:
<code>\IfPackageLoadedTF</code>	13	<code>\c_empty_tl</code>
<code>\iftrue</code>	6	<code>\tl_gset:Nn</code>
<code>\in@</code>	11	<code>\tl_new:N</code>
<code>\let</code>	11	<code>\tl_set:Nn</code>
<code>\long</code>	13	<code>\tl_set_eq:NN</code>
<code>\message</code>	10	<code>\tl_use:N</code>
<code>\special</code>	2	<code>\l_tmpa_tl</code>
<code>\string</code>	11	token commands:
		<code>\token_to_str:N</code>