IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

Section 3: Shell Command Language

The shell is a command language interpreter. This section describes the syntax of that command language as it is used by the `sh` utility and the functions in 7.1 [such as *system* () and *popen* () in the C binding].

The shell operates according to the following general overview of operations. The specific details are included in the cited clauses and subclauses of this section. The shell:

(1) Reads its input from a file (see `sh` in 4.56), from the `-c` option, or from one of the functions in 7.1. If the first line of a file of shell commands starts with the characters `#!`, the results are unspecified.

(2) Breaks the input into tokens: words and operators. (See 3.3.)

(3) Parses the input into simple (3.9.1) and compound (3.9.4) commands.

(4) Performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and fields to be treated as a command and arguments (3.6).

(5) Performs redirection (3.7) and removes redirection operators and their operands from the parameter list.

(6) Executes a function (3.9.5), built-in (3.14), executable file, or script, giving the name of the command (or, in the case of a function within a script, the name of the script) as the ''zero'th'' argument and the remaining words and fields as parameters (3.9.1.1).

(7) Optionally waits for the command to complete and collects the exit status (3.8.2).

3.0.1 Shell Command Language Rationale. *(This subclause is not a part of P1003.2)*

The System V shell was selected as the starting point for this standard. The BSD C-shell was excluded from consideration, for the following reasons:

(1) Most historically portable shell scripts assume the Version 7 ''Bourne'' shell, from which the System V shell is derived.

(2) The majority of tutorial materials on shell programming assume the System V shell.

Despite the selection of the System V shell, the developers of the standard did not limit the possibilities for a shell command language that was upward-compatible.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

The only programmatic interfaces to the shell language are through the functions in 7.1 and the `sh` utility. Most implementations provide an interface to, and processing mode for, the shell that is suitable for direct user interaction. The behavior of this interactive mode is not defined by this standard; however, places where historically an interactive shell behaves differently from the behavior described here are noted.

(1) Aliases are not included in the base POSIX. 2 because they duplicate functionality already available to applications with functions. In early drafts, the search order of simple command lookup was ‘‘aliases, built-ins, functions, file system,’’ and therefore an alias was necessary to create a user-defined command having the same name as a built-in. To retain this capability, the search order has changed to ‘‘special built-ins, functions, built-ins, file system,’’ and a built-in, called `command`, has been added, which disables the looking up of functions. Aliases are a part of the POSIX. 2a UPE because they are widely used by human users, as differentiated from applications.

(2) All references to job control and related commands have been omitted from the base POSIX. 2. POSIX. 2 describes the noninteractive operation of the shell; job control is outside the scope of this standard until the UPE revision is developed. Apparently it is not widely known that traditionally, even in a job control environment, the commands executed during the execution of a shell script are not placed into separate process groups. If they were, one could not stop the execution of the shell script from the interactive shell, for example. This standard does not require or prohibit job control; it simply does not mention it.

(3) The conditional command (double bracket `[[ ]]`) was removed from an earlier draft. Objections were lodged that the real problem is misuse of the `test` command (`[`), and putting it into the shell is the wrong way to fix the problem. Instead, proper documentation and a new shell reserved word (`!`) are sufficient. Tests that require multiple `test` operations can be done at the shell level using individual invocations of the `test` command and shell logicals, rather than the error prone `–o` flag of `test`.

(4) Exportable functions were removed from an earlier draft. See the rationale in 3.9.5.1.

The construct #! is reserved for implementations wishing to provide that extension. If it were not reserved, the standard would disallow it by forcing it to be a comment. As it stands, a conforming application shall not use #! as the first line of a shell script.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.1 Shell Definitions

The following terms are used in Section 3. Because they are specific to the shell, they do not appear in 2.2.2.

3.1.1 control operator: A token that performs a control function.

It is one of the following symbols:

&	)	<newline>
&&	;	|
(	;;	||

The end-of-input indicator used internally by the shell is also considered a control operator. See 3.3.

On some systems, the symbol ((is a control operator; its use produces unspecified results. 1

3.1.2 expand: When not qualified, the act of applying all the expansions described in 3.6.

3.1.3 field: A unit of text that is the result of parameter expansion (3.6.2), arithmetic expansion (3.6.4), command substitution (3.6.3), or field splitting (3.6.5).

During command processing (see 3.9.1), the resulting fields are used as the command name and its arguments.

3.1.4 interactive shell: A processing mode of the shell that is suitable for direct user interaction.

The behavior in this mode is not defined by this standard.

NOTE: The preceding sentence is expected to change following the eventual approval of the UPE supplement.

3.1.5 name: A word consisting solely of underscores, digits, and alphabetics from the portable character set (see 2.4).

The first character of a name shall not be a digit.

3.1.6 operator: Either a control operator or a redirection operator.

3.1.7 parameter: An entity that stores values.

There are three types of parameters: variables (named parameters), positional parameters, and special parameters. Parameter expansion is accomplished by introducing a parameter with the $ character. See 3.5.

3.1.8 positional parameter: A parameter denoted by a single digit or one or more digits in curly braces.

See 3.5.1.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.1.9 redirection: A method of associating files with the input/output of commands.

See 3.7.

3.1.10 redirection operator: A token that performs a redirection function.

It is one of the following symbols:

< > >| << >> <& >& <<- <>

3.1.11 special parameter: A parameter named by a single character from the following list:

* @ # ? ! – $ 0

See 3.5.2.

3.1.12 subshell: A shell execution environment, distinguished from the main or current shell execution environment by the attributes described in 3.12.

3.1.13 token: A sequence of characters that the shell considers as a single unit when reading input, according to the rules in 3.3.

A token is either an operator or a word.

3.1.14 variable: A named parameter. See 3.5.

3.1.15 variable assignment [assignment]: A word consisting of the following parts

varname =*value*

When used in a context where assignment is defined to occur (see 3.9.1) and at no other time, the *value* (representing a word or field) shall be assigned as the value of the variable denoted by *varname* . The *varname* and *value* parts meet the requirements for a name and a word, respectively, except that they are delimited by the embedded unquoted equals-sign in addition to the delimiting described in 3.3. In all cases, the variable shall be created if it did not already exist. If *value* is not specified, the variable shall be given a null value.

An alternative form of variable assignment:

symbol =*value*

(where *symbol* is a valid word delimited by an equals-sign, but not a valid name) produces unspecified results.

3.1.16 word: A token other than an operator.

In some cases a word is also a portion of a word token: in the various forms of parameter expansion (3.6.2), such as ${*name* –*word* }, and variable assignment, such as *name*=*word* , the word is the portion of the token depicted by *word* . The concept of a word is no longer applicable following word expansions—only fields remain; see 3.6.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.1.17 Shell Definitions Rationale. *(This subclause is not a part of P1003.2)*

The *word* =*word* form of variable assignment was included, producing unspecified results, to allow the KornShell *name* [*expression*]=*value* syntax to conform.

The `((` symbol is a control operator in the KornShell, used for an alternative syntax of an arithmetic expression command. A strictly conforming POSIX.2 application cannot use `((` as a single token [with the obvious exception of the `$((` form described in POSIX.2]. The decision to require this is based solely on the pragmatic knowledge that there are many more historical shell scripts using the KornShell syntax than there might be using nested subshells, such as

```
((foo))        or        ((foo);(bar))
```

The latter example should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. Thus, in most cases, while a few scripts will no longer be strictly portable, the chances of breaking existing scripts is even smaller.

There are no explicit limits in this standard on the sizes of names, words, lines, or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a byte at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice such as makefiles, where `make` removes the `<newline>`s associated with the commands for a target and presents the shell with one very long line. The text in 2.11.5.2 does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

3.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to preserve the literal meaning of the special characters in the next paragraph; prevent reserved words from being recognized as such; and prevent parameter expansion and command substitution within here-document processing (see 3.7.4).

The following characters shall be quoted if they are to represent themselves:

```
|   &   ;   <   >   (   )   $   ‘   \   "   ’
<space>   <tab>   <newline>
```

and the following may need to be quoted under certain circumstances. That is, these characters may be special depending on conditions described elsewhere in the standard:

```
*   ?   [   #   ~   =   %
```

The various quoting mechanisms are the escape character, single-quotes, and double-quotes. The here-document represents another form of quoting; see 3.7.4.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.2.1 Escape Character (Backslash)

A backslash that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the backslash, the shell shall interpret this as line continuation. The backslash and <newline> shall be removed before splitting the input into tokens.

3.2.2 Single-Quotes

Enclosing characters in single-quotes (' ') shall preserve the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

3.2.3 Double-Quotes

Enclosing characters in double-quotes (" ") shall preserve the literal value of all characters within the double-quotes, with the exception of the characters dollar-sign, backquote, and backslash, as follows:

$ The dollar-sign shall retain its special meaning introducing parameter expansion (see 3.6.2), a form of command substitution (see 3.6.3), and arithmetic expansion (see 3.6.4).

The input characters within the quoted string that are also enclosed between $(and the matching) shall not be affected by the double-quotes, but rather shall define that command whose output replaces the $(...) when the word is expanded. The tokenizing rules in 3.3 shall be applied recursively to find the matching).

Within the string of characters from an enclosed ${ to the matching }, an even number of unescaped double-quotes or single-quotes, if any, shall occur. A preceding backslash character shall be used to escape a literal { or }. The rule in 3.6.2 shall be used to determine the matching }.

` The backquote shall retain its special meaning introducing the other form of command substitution (see 3.6.3). The portion of the quoted string from the initial backquote and the characters up to the next backquote that is not preceded by a backslash, having escape characters removed, defines that command whose output replaces `...` when the word is expanded. Either of the following cases produces undefined results:

— A single- or double-quoted string that begins, but does not end, within the `...` sequence.

— A `...` sequence that begins, but does not end, within the same double-quoted string.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

\ The backslash shall retain its special meaning as an escape character (see 3.2.1) only when followed by one of the characters:

```
$ ' " \ <newline>
```

A double-quote shall be preceded by a backslash to be included within double-quotes. The parameter @ has special meaning inside double-quotes and is described in 3.5.2.

3.2.4 Quotes Rationale. *(This subclause is not a part of P1003.2)*

A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded quote can be created by writing, for example, `'a'\''b'`, which yields `a'b`. (See 3.6.5 for a better understanding of how portions of words are either split into fields or remain concatenated.) A single token can be made up of concatenated partial strings containing all three kinds of quoting/escaping, thus permitting any combination of characters.

The escaped `<newline>` used for line continuation is removed entirely from the input and is not replaced by any white space. Therefore, it cannot serve as a token separator.

In double-quoting, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unmodified and the character immediately following it is also left unmodified. Thus, for example:

```
"\$"  ⇒  $
"\a"  ⇒  \a
```

It would be desirable to include the statement ‘‘The characters from an enclosed `${` to the matching `}` shall not be affected by the double-quotes,’’ similar to the one for `$( )`. However, historical practice in the System V shell prevents this. The requirement that double-quotes be matched inside `${...}` within double-quotes and the rule for finding the matching `}` in 3.6.2 eliminate several subtle inconsistencies in expansion for historical shells in rare cases; for example,

```
"${foo-bar"}
```

yields `bar` when `foo` is not defined, and is an invalid substitution when `foo` is defined, in many historical shells. The differences in processing the `"${...}"` form have led to inconsistencies between the historical System V, BSD, and KornShells, and the text in POSIX.2 is an attempt to converge them without breaking many applications. A consequence of the new rule is that single-quotes cannot be used to quote the `}` within `"${...}"`; for example

```
unset bar
foo="${bar-'}'}"
```

is invalid because the `"${...}"` substitution contains an unpaired unescaped single-quote. The backslash can be used to escape the `}` in this example to

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

achieve the desired result:

```
unset bar
foo="${bar-\}}"
```

The only alternative to this compromise between shells would be to make the behavior unspecified whenever the literal characters ', {, }, and " appear within ${...}. To write a portable script that uses these values, a user would have to assign variables, say,

```
squote=\' dquote=\" lbrace='{' rbrace='}'
${foo-$squote$rbrace$squote}
```

rather than

```
${foo-"'}'"}
```

Some systems have allowed the end of the word to terminate the backquoted command substitution, such as in

```
"`echo hello"
```

This usage is undefined in POSIX. 2, where the matching backquote is required. The other undefined usage can be illustrated by the example:

```
sh -c `` echo "foo``
```

The description of the recursive actions involving command substitution can be illustrated with an example. Upon recognizing the introduction of command substitution, the shell must parse input (in a new context), gathering the ''source'' for the command substitution until an unbalanced) or ` is located. For example, in the following

```
echo "$(date; echo "
        one" )"
```

the double-quote following the `echo` does not terminate the first double-quote; it is part of the command substitution ''script.'' Similarly, in

```
echo "$(echo *)"
```

the asterisk is not quoted since it is inside command substitution; however,

```
echo "$(echo "*")"
```

is quoted (and represents the asterisk character itself).

3.3 Token Recognition

The shell reads its input in terms of lines from a file, from a terminal in the case of an interactive shell, or from a string in the case of `sh -c` or *system* (). The input lines can be of unlimited length. These lines are parsed using two major modes: ordinary token recognition and processing of here-documents.

When an `io_here` token has been recognized by the grammar (see 3.10), one or more of the immediately subsequent lines form the body of one or more here-

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

documents and shall be parsed according to the rules of 3.7.4.

When it is not processing an `io_here`, the shell shall break its input into tokens by applying the first applicable rule below to the next character in its input. The token shall be from the current position in the input until a token is delimited according to one of the rules below; the characters forming the token are exactly those in the input, including any quoting characters. If it is indicated that a token is delimited, and no characters have been included in a token, processing shall continue until an actual token is delimited.

(1) If the end of input is recognized, the current token shall be delimited. If there is no current token, the end-of-input indicator shall be returned as the token.

(2) If the previous character was used as part of an operator and the current character is not quoted and can be used with the current characters to form an operator, it shall be used as part of that (operator) token.

(3) If the previous character was used as part of an operator and the current character cannot be used with the current characters to form an operator, the operator containing the previous character shall be delimited.

(4) If the current character is backslash, single-quote, or double-quote (`\`, `'`, or `"`) and it is not quoted, it shall affect quoting for subsequent character(s) up to the end of the quoted text. The rules for quoting are as described in 3.2. During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for `<newline>` joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the quote mark and the end of the quoted text. The token shall not be delimited by the end of the quoted field.

(5) If the current character is an unquoted `$` or `‘`, the shell shall identify the start of any candidates for parameter expansion (3.6.2), command substitution (3.6.3), or arithmetic expansion (3.6.4) from their introductory unquoted character sequences: `$` or `${`, `$(` or `‘`, and `$((`, respectively. The shell shall read sufficient input to determine the end of the unit to be expanded (as explained in the cited subclauses). While processing the characters, if instances of expansions or quoting are found nested within the substitution, the shell shall recursively process them in the manner specified for the construct that is found. The characters found from the beginning of the substitution to its end, allowing for any recursion necessary to recognize embedded constructs, shall be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token shall not be delimited by the end of the substitution.

(6) If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) shall be delimited. The current character shall be used as the beginning of the next (operator) token.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(7) If the current character is an unquoted <newline>, the current token shall be delimited.

(8) If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character is discarded.

(9) If the previous character was part of a word, the current character is appended to that word.

(10) If the current character is a #, it and all subsequent characters up to, but excluding, the next <newline> are discarded as a comment. The <newline> that ends the line is not considered part of the comment.

(11) The current character is used as the start of a new word.

Once a token is delimited, it shall be categorized as required by the grammar in 3.10.

3.3.1 Token Recognition Rationale. *(This subclause is not a part of P1003.2)*

The (3) rule about combining characters to form operators is not meant to preclude systems from extending the shell language when characters are combined in otherwise invalid ways. Portable applications cannot use invalid combinations and test suites should not penalize systems that take advantage of this fact. For example, the unquoted combination |& is not valid in a POSIX.2 script, but has a specific KornShell meaning.

The (10) rule about # as the current character is the first in the sequence in which a new token is being assembled. The # starts a comment only when it is at the beginning of a token. This rule is also written to indicate that the search for the end-of-comment does not consider escaped <newline> specially, so that a comment cannot be continued to the next line.

3.4 Reserved Words

Reserved words are words that have special meaning to the shell. (See 3.9.) The following words shall be recognized as reserved words:

```
!       elif    fi      in      while
case    else    for     then    {[4)]
do      esac    if      until   }
done
```

4) In some historical systems, the curly braces are treated as control operators. To assist in future standardization activities, portable applications should avoid using unquoted braces to represent the characters themselves. It is possible that a future version of POSIX.2 may require this, although probably not for the often-used `find {}` construct.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

This recognition shall occur only when none of the characters are quoted and when the word is used as:

(1) The first word of a command

(2) The first word following one of the reserved words other than `case`, `for`, or `in`

(3) The third word in a `case` or `for` command (only `in` is valid in this case)

See the grammar in 3.10.

The following words may be recognized as reserved words on some systems (when none of the characters are quoted), causing unspecified results: 2

```
function   select     [[         ]]
```

Words that are the concatenation of a name and a colon (`:`) are reserved; their use produces unspecified results.

3.4.1 Reserved Words Rationale. *(This subclause is not a part of P1003.2)*

All reserved words are recognized syntactically as such in the contexts described. However, it is useful to point out that `in` is the only meaningful reserved word after a `case` or `for`; similarly, `in` is not meaningful as the first word of a simple command.

Reserved words are recognized only when they are delimited (i.e., meet the definition of *word* ; see 3.1.16), whereas operators are themselves delimiters. For instance, `(` and `)` are control operators, so that no <space> is needed in `(list)`. However, `{` and `}` are reserved words in `{ list;}`, so that in this case the leading <space> and semicolon are required.

The list of unspecified reserved words is from the KornShell, so portable applications cannot use them in places a reserved word would be recognized. This list 2
contained `time` in earlier drafts, but it was removed when the `time` utility was 2
selected for the UPE. 2

There was a strong argument for promoting braces to operators (instead of reserved words), so they would be syntactically equivalent to subshell operators. Concerns about compatibility outweighed the advantages of this approach. Nevertheless, portable applications should consider quoting `{` and `}` when they represent themselves.

The restriction on ending a name with a colon is to allow future implementations that support named labels for flow control. See the rationale for `break` (3.14.1.1).

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.5 Parameters and Variables

A parameter can be denoted by a name, a number, or one of the special characters listed in 3.5.2. A variable is a parameter denoted by a name.

A parameter is set if it has an assigned value (null is a valid value). Once a variable is set, it can only be unset by using the `unset` special built-in command.

3.5.1 Positional Parameters

A positional parameter is a parameter denoted by the decimal value represented by one or more digits, other than the single digit 0. When a positional parameter with more than one digit is specified, the application shall enclose the digits in braces (see 3.6.2). Positional parameters are initially assigned when the shell is invoked (see `sh` in 4.56), temporarily replaced when a shell function is invoked (see 3.9.5), and can be reassigned with the `set` special built-in command.

3.5.1.1 Positional Parameters Rationale. *(This subclause is not a part of P1003.2)*

The digits denoting the positional parameters are always interpreted as a decimal value, even if there is a leading zero.

3.5.2 Special Parameters

Listed below are the special parameters and the values to which they shall expand. Only the values of the special parameters are listed; see 3.6 for a detailed summary of all the stages involved in expanding words.

`*` Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string (see 3.2.3), it expands to a single field with the value of each parameter separated by the first character of the **IFS** variable, or by a `<space>` if **IFS** is unset.

`@` Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, each positional parameter expands as a separate field, with the provision that the expansion of the first parameter is still joined with the beginning part of the original word (assuming that the expanded parameter was embedded within a word), and the expansion of the last parameter is still joined with the last part of the original word. If there are no positional parameters, the expansion of `@` shall generate zero fields, even when `@` is double-quoted.

`#` Expands to the decimal number of positional parameters.

`?` Expands to the decimal exit status of the most recent pipeline (see 3.9.2).

`-` (Hyphen) Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation, by the `set` special built-in command, or implicitly by the shell.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

443 `$` Expands to the decimal process ID of the invoked shell. In a subshell
444 (see 3.12), `$` shall expand to the same value as that of the current shell.

445 `!` Expands to the decimal process ID of the most recent background com-
446 mand (see 3.9.3) executed from the current shell. For a pipeline, the 1
447 process ID is that of the last command in the pipeline.

448 `0` (Zero.) Expands to the name of the shell or shell script. See `sh` (4.56)
449 for a detailed description of how this name is derived.

450 See the description of the **IFS** variable in 3.5.3.

451 **3.5.2.1 Special Parameters Rationale.** *(This subclause is not a part of P1003.2)*

452 Most historical implementations implement subshells by forking; thus, the special
453 parameter `$` does not necessarily represent the process ID of the shell process exe-
454 cuting the commands since the subshell execution environment preserves the
455 value of `$`.

456 If a subshell were to execute a background command, the value of its parent's `$!` 1
457 would not change. For example: 1

```
458 (                                                                    1
459 date &                                                              1
460 echo $!                                                            1
461 )                                                                   1
462 echo $!                                                            1
```

463 would echo two different values for `$!`. 1

464 The descriptions of parameters * and @ assume the reader is familiar with the
465 field splitting discussion in 3.6.5 and understands that portions of the word will
466 remain concatenated unless there is some reason to split them into separate
467 fields. Some examples of the * and @ properties, including the concatenation
468 aspects:

```
469 set "abc" "def ghi" "jkl"
470 echo $*          => "abc" "def" "ghi" "jkl"
471 echo "$*"        => "abc def ghi jkl"
472 echo $@          => "abc" "def" "ghi" "jkl"
```

473 *but*

```
474 echo "$@"        => "abc" "def ghi" "jkl"
475 echo "xx$@yy"    => "xxabc" "def ghi" "jklyy"
476 echo "$@$@"      => "abc" "def ghi" "jklabc" "def ghi" "jkl"
```

477 In the preceding examples, the double-quote characters that appear after the `=>`
478 do not appear in the output and are used only to illustrate word boundaries.

479 Historical versions of the Bourne shell have used `<space>` as a separator
480 between the expanded members of `"$*"`. The KornShell has used the first char-
481 acter in **IFS**, which is `<space>` by default. If **IFS** is set to a null string, this is not 1
482 equivalent to unsetting it; its first character will not exist, so the parameter 1
483 values are concatenated. For example: 1

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

The `$-` can be used to save and restore `set` options:

```
Save=$(echo $- | sed 's/[ics]//g')
...
set +aCefnuvx
set -$Save
```

The three options are removed using `sed` in the example because they may appear in the value of `$-` (from the `sh` command line), but are not valid options to `set`.

The command name (parameter `0`) is not counted in the number given by # because it is a special parameter, not a positional parameter.

3.5.3 Variables

Variables shall be initialized from the environment (as defined by POSIX.1 {8}) and can be given new values with variable assignment commands. If a variable is initialized from the environment, it shall be marked for export immediately; see 3.14.8. New variables can be defined and initialized with variable assignments, with the `read` or `getopts` utilities, with the *name* parameter in a `for` loop (see 3.9.4.2), with the `${`*name* `=`*word* `}` expansion, or with other mechanisms provided as implementation extensions. The following variables shall affect the execution of the shell:

HOME This variable shall be interpreted as the pathname of the user's home directory. The contents of **HOME** are used in Tilde Expansion (see 3.6.1).

IFS *Input field separators* : a string treated as a list of characters that is used for field splitting and to split lines into fields with the `read` command. If **IFS** is not set, the shell shall behave as if the value of **IFS** were the `<space>`, `<tab>`, and `<newline>` characters. (See 3.6.5.)

LANG This variable shall provide a default value for the **LC_*** variables, as described in 2.6.

LC_ALL This variable shall interact with the **LANG** and **LC_*** variables as described in 2.6.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

LC_COLLATE This variable shall determine the behavior of range expressions, equivalence classes, and multicharacter collating elements within pattern matching.

LC_CTYPE This variable shall determine the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters), which characters are defined as letters (character class `alpha`), and the behavior of character classes within pattern matching.

LC_MESSAGES This variable shall determine the language in which messages should be written.

PATH This variable represents a string formatted as described in 2.6, used to effect command interpretation. See 3.9.1.1.

3.5.3.1 Variables Rationale. *(This subclause is not a part of P1003.2)*

A description of **PWD** (which is automatically set by the KornShell whenever the current working directory changes) was omitted because its functionality is easily reproduced using `$(pwd)`.

See the discussion of **IFS** in 3.6.5.1.

Other common environment variables used by historical shells are not specified by this standard, but they should be reserved for the historical uses. For interactive use, other shell variables are expected to be introduced by the UPE (and this rationale will be updated accordingly): **ENV** , **FCEDIT** , **HISTFILE** , **HISTSIZE** , **LINENO** , **PPID** , **PS 1**, **PS 2**, **PS 4**.

Tilde expansion for components of the **PATH** in an assignment such as:

```
PATH=~hlj/bin:~dwc/bin:$PATH
```

is a feature of some historical shells and is allowed by the wording of 3.6.1. Note that the tildes are expanded during the assignment to **PATH** , not when **PATH** is accessed during command search.

3.6 Word Expansions

This clause describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained in the following subclauses.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter @ within double-quotes, as is described in 3.5.2.

The order of word expansion shall be as follows:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

(1) Tilde Expansion (see 3.6.1), Parameter Expansion (see 3.6.2), Command Substitution (see 3.6.3), and Arithmetic Expansion (see 3.6.4) shall be performed, beginning to end. [See item (5) in 3.3.]

(2) Field Splitting (see 3.6.5) shall be performed on fields generated by step (1) unless **IFS** is null.

(3) Pathname Expansion (see 3.6.6) shall be performed, unless `set –f` is in effect.

(4) Quote Removal (see 3.6.7) shall always be performed last.

The expansions described in this clause shall occur in the same shell environment as that in which the command is executed.

If the complete expansion appropriate for a word results in an empty field, that empty field shall be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The `$` character is used to introduce parameter expansion, command substitution, or arithmetic evaluation. If an unquoted `$` is followed by a character that is either not numeric, the name of one of the special parameters (see 3.5.2), a valid first character of a variable name, a left curly brace (`{`), or a left parenthesis, the result is unspecified.

3.6.0.1 Word Expansions Rationale. *(This subclause is not a part of P1003.2)*

IFS is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behavior. All useful applications of **IFS** use it for parsing input of the `read` utility and for splitting the results of parameter and command substitution. New versions of the shell have fixed this bug, and POSIX. 2 requires the corrected behavior.

The rule concerning expansion to a single field requires that if `foo=abc` and `bar=def`, that

```
"$foo""$bar"
```

expands to the single field

```
abcdef
```

The rule concerning empty fields can be illustrated by:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
$ unset foo
$ set $foo bar '' xyz "$foo" abc
$ for i
> do
>       echo "-$i-"
> done
-bar-
--
-xyz-
--
-abc-
```

Step (1) indicates that Tilde Expansion, Parameter Expansion, Command Substitution, and Arithmetic Expansion are all processed simultaneously as they are scanned. For example, the following is valid arithmetic:

```
x=1
echo $(( $(echo 3)+$x ))
```

An earlier draft stated that Tilde Expansion preceded the other steps, but this is not the case in known historical implementations; if it were, and a referenced home directory contained a $ character, expansions would result within the directory name.

3.6.1 Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, followed by all of the characters preceding the first unquoted slash in the word, or all the characters in the word if there is no slash. In an assignment (see 3.1.15), multiple tilde prefixes can be used: at the beginning of the word (i.e., following the equals-sign of the assignment) and/or following any unquoted colon. A tilde prefix in an assignment is terminated by the first unquoted colon or slash. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde shall be treated as a possible login name from the user database (see POSIX.1 {8} Section 9). A portable login name cannot contain characters outside the set given in the description of the **LOGNAME** environment variable in POSIX.1 {8}. If the login name is null (i.e., the tilde-prefix contains only the tilde), the tilde-prefix shall be replaced by the value of the variable **HOME**. If **HOME** is unset, the results are unspecified. Otherwise, the tilde-prefix shall be replaced by a pathname of the home directory associated with the login name obtained using the equivalent of the POSIX.1 {8} *getpwnam* () function. If the system does not recognize the login name, the results are undefined.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.6.1.1 Tilde Expansion Rationale. *(This subclause is not a part of P1003.2)*

The text about quoting of the word indicates that `\~hlj/`, `~h\lj/`, `~"hlj"/`, `~hlj\/`, and `~hlj/` are not equivalent: only the last will cause tilde expansion.

Tilde expansion generally occurs only at the beginning of words, but POSIX. 2 has adopted an exception based on historical practice in the KornShell:

```
PATH=/posix/bin:~dgk/bin
```

is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behavior because any of the following are reasonable substitutes:

```
PATH=$(printf %s: ~rms/bin ~bfox/bin ...)
PATH=$(printf %s ~karels/bin : ~bostic/bin)
for Dir in ~maart/bin ~srb/bin ...
do
        PATH=${PATH:+$PATH:}$Dir
done
```

(In the first command, any number of directory names are concatenated and separated with colons, but it may be undesirable to end the variable with a colon because this is an obsolescent means to include dot at the end of the **PATH**. In the second, explicit colons are used for each directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.)

The exception was included to avoid breaking numerous KornShell scripts and interactive users and despite the fact that variable assignments in scripts derived from other systems will have to use quoting in some cases to allow literal tildes in strings. (This latter problem should be relatively rare because only tildes preceding known login names in unquoted strings are affected.)

Note that expressions in operands such as

```
make -k mumble LIBDIR=~chet/lib
```

do not qualify as shell variable assignments and tilde expansion is not performed (unless the command does so itself, which `make` does not).

In an earlier draft, tilde expansion occurred following any unquoted equals-sign or colon, but this was removed because of its complexity and to avoid breaking commands such as:

```
rcp hostname:~marc/.profile .
```

A suggestion was made that the special sequence ‘$~’’ should be allowed to force tilde expansion anywhere. Since this is not historical practice, it has been left for future implementations to evaluate. (The description in 3.2 requires that a dollar-sign be quoted to represent itself, so the $~ combination is already unspecified.)

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

The results of giving tilde with an unknown login name are undefined because the KornShell ~+ and ~− constructs make use of this condition, but in general it is an error to give an incorrect login name with tilde. The results of having **HOME** unset are unspecified because some historical shells treat this as an error.

3.6.2 Parameter Expansion

The format for parameter expansion is as follows:

```
${expression }
```

where *expression* consists of all characters until the matching `}`. Any `}` escaped by a backslash or within a quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, shall not be examined in determining the matching `}`.

The simplest form for parameter expansion is:

```
${parameter }
```

The value, if any, of *parameter* shall be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when *parameter* is followed by a character that could be interpreted as part of the name. The matching closing brace shall be determined by counting brace levels, skipping over enclosed quoted strings and command substitutions.

If the parameter name or symbol is not enclosed in braces, the expansion shall use the longest valid name (see 3.1.5), whether or not the symbol represented by that name exists. If a parameter expansion occurs inside double-quotes:

— Pathname expansion shall not be performed on the results of the expansion.

— Field splitting shall not be performed on the results of the expansion, with the exception of @; see 3.5.2.

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of *parameter*, as described below), *word* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *word* is not needed, it shall not be expanded. The `}` character that delimits the following parameter expansion modifications shall be determined as described previously in this subclause and in 3.2.3. (For example, `${foo-bar}xyz}` would result in the expansion of `foo` followed by the string `xyz}` if `foo` is set, else the string `barxyz}`).

`${parameter :-word }` **Use Default Values.** If *parameter* is unset or null, the expansion of *word* shall be substituted; otherwise, the value of *parameter* shall be substituted.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

709 `${`*parameter* `:=`*word* `}` **Assign Default Values.** If *parameter* is unset or
710 null, the expansion of *word* shall be assigned to
711 *parameter* . In all cases, the final value of *parame-*
712 *ter* shall be substituted. Only variables, not posi-
713 tional parameters or special parameters, can be
714 assigned in this way.

715 `${`*parameter* `:?[`*word* `]}` **Indicate Error if Null or Unset.** If *parameter* is
716 unset or null, the expansion of *word* (or a message
717 indicating it is unset if *word* is omitted) shall be
718 written to standard error and the shell shall exit
719 with a nonzero exit status. Otherwise, the value of
720 *parameter* shall be substituted. An interactive shell
721 need not exit.

722 `${`*parameter* `:+`*word* `}` **Use Alternate Value.** If *parameter* is unset or
723 null, null shall be substituted; otherwise, the
724 expansion of *word* shall be substituted.

725 In the parameter expansions shown previously, use of the colon in the format
726 results in a test for a parameter that is unset or null; omission of the colon results
727 in a test for a parameter that is only unset.

728 `${#`*parameter* `}` **String Length.** The length in characters of the
729 value of *parameter* . If *parameter* is * or @, the
730 result of the expansion is unspecified.

731 The following four varieties of parameter expansion provide for substring process-
732 ing. In each case, pattern matching notation (see 3.13), rather than regular
733 expression notation, shall be used to evaluate the patterns. If *parameter* is * or @,
734 the result of the expansion is unspecified. Enclosing the full parameter expansion
735 string in double-quotes shall not cause the following four varieties of pattern char- 1
736 acters to be quoted, whereas quoting characters within the braces shall have this
737 effect.

738 `${`*parameter* `%`*word* `}` **Remove Smallest Suffix Pattern.** The *word*
739 shall be expanded to produce a pattern. The
740 parameter expansion then shall result in *parame-*
741 *ter* , with the smallest portion of the suffix matched
742 by the *pattern* deleted.

743 `${`*parameter* `%%`*word* `}` **Remove Largest Suffix Pattern.** The *word* shall
744 be expanded to produce a pattern. The parameter
745 expansion then shall result in *parameter* , with the
746 largest portion of the suffix matched by the *pattern*
747 deleted.

748 `${`*parameter* `#`*word* `}` **Remove Smallest Prefix Pattern.** The *word*
749 shall be expanded to produce a pattern. The
750 parameter expansion then shall result in *parame-*
751 *ter* , with the smallest portion of the prefix matched
752 by the *pattern* deleted.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.