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All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change.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]

753 `${`*parameter* `##`*word* `}` **Remove Largest Prefix Pattern.** The *word* shall
754 be expanded to produce a pattern. The parameter
755 expansion then shall result in *parameter* , with the
756 largest portion of the prefix matched by the *pattern*
757 deleted.

758 **3.6.2.1 Parameter Expansion Rationale.** *(This subclause is not a part of P1003.2)*

759 When the shell is scanning its input to determine the boundaries of a name, it is
760 not bound by its knowledge of what names are already defined. For example, if `F`
761 is a defined shell variable, the command `"echo $Fred"` does not echo the value
762 of `$F` followed by `red`; it selects the longest possible valid name, `Fred`, which in
763 this case might be unset.

764 The rule for finding the closing `}` in `${...}` is the one used in the KornShell and
765 is upward compatible with the Bourne shell, which does not determine the closing
766 `}` until the word is expanded. The advantage of this is that incomplete expan-
767 sions, such as

768 `${foo`

769 can be determined during tokenization, rather than during expansion.

770 The four expansions with the optional colon have been hard to understand from
771 the historical documentation. The following table summarizes the effect of the
772 colon:

	parameter **set and not null**	***parameter*** **set but null**	***parameter*** **unset**
`${`*parameter* `:-`*word* `}`	substitute *parameter*	substitute *word*	substitute *word*
`${`*parameter* `-`*word* `}`	substitute *parameter*	substitute null	substitute *word*
`${`*parameter* `:=`*word* `}`	substitute *parameter*	assign *word*	assign *word*
`${`*parameter* `=`*word* `}`	substitute *parameter*	substitute *parameter*	assign *word*
`${`*parameter* `:?`*word* `}`	substitute *parameter*	error, exit	error, exit
`${`*parameter* `?`*word* `}`	substitute *parameter*	substitute null	error, exit
`${`*parameter* `:+`*word* `}`	substitute *word*	substitute null	substitute null
`${`*parameter* `+`*word* `}`	substitute *word*	substitute *word*	substitute null

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

In all cases shown with ‘‘substitute,’’ the expression is replaced with the value shown. In all cases shown with ‘‘assign,’’ *parameter* is assigned that value, which also replaces the expression.

The string length and substring capabilities were included because of the demonstrated need for them, based on their usage in other shells, such as C-shell and KornShell.

Historical versions of the KornShell have not performed tilde expansion on the word part of parameter expansion; however, it is more consistent to do so.

Examples

`${`*parameter* `:–`*word* `}`

In this example, `ls` is executed only if `x` is null or unset. [The `$(ls)` command substitution notation is explained in 3.6.3.]

```
${x:-$(ls)}
```

`${`*parameter* `:=`*word* `}`

```
unset X
echo ${X:=abc}
abc
```

`${`*parameter* `:?`*word* `}`

```
unset posix
echo ${posix:?}
sh: posix: parameter null or not set
```

`${`*parameter* `:+`*word* `}`

```
set a b c
echo ${3:+posix}
posix
```

`${#`*parameter* `}`

```
HOME=/usr/posix
echo ${#HOME}
10
```

`${`*parameter* `%`*word* `}`

```
x=file.c
echo ${x%.c}.o
file.o
```

`${`*parameter* `%%`*word* `}`

```
x=posix/src/std
echo ${x%%/*}
posix
```

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

828 `${`*parameter* `#`*word* `}`

```
829             x=$HOME/src/cmd
830             echo ${x#$HOME}
831             /src/cmd
```

832 `${`*parameter* `##`*word* `}`

```
833             x=/one/two/three
834             echo ${x##*/}
835             three
```

836 The double-quoting of patterns is different depending on where the double-quotes
837 are placed:

838 `"${x#*}"` The asterisk is a pattern character.

839 `${x#"*"}` The literal asterisk is quoted and not special.

840 ### 3.6.3 Command Substitution

841 Command substitution allows the output of a command to be substituted in place
842 of the command name itself. Command substitution shall occur when the com-
843 mand is enclosed as follows:

844 `$(`*command* `)`

845 or (‘‘backquoted’’ version):

846 `‘`*command* `‘`

847 The shell shall expand the command substitution by executing *command* in a sub-
848 shell environment (see 3.12) and replacing the command substitution [the text of
849 *command* plus the enclosing `$( )` or backquotes] with the standard output of the
850 command, removing sequences of one or more `<newline>`s at the end of the sub-
851 stitution. (Embedded `<newline>`s before the end of the output shall not be
852 removed; however, during field splitting, they may be translated into `<space>`s,
853 depending on the value of **IFS** and quoting that is in effect.)

854 Within the backquoted style of command substitution, backslash shall retain its
855 literal meaning, except when followed by

856 `$ ‘ \`

857 (dollar-sign, backquote, backslash). The search for the matching backquote shall 2
858 be satisfied by the first backquote found without a preceding backslash; during 2
859 this search, if a nonescaped backquote is encountered within a shell comment, a 2
860 here-document, an embedded command substitution of the `$(`*command* `)` form, or 2
861 a quoted string, undefined results occur. A single- or double-quoted string that
862 begins, but does not end, within the `‘...‘` sequence produces undefined results.

863 With the `$(`*command* `)` form, all characters following the open parenthesis to the
864 matching closing parenthesis constitute the *command* . Any valid shell script can 2
865 be used for *command* , except: 2

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

— A script consisting solely of redirections produces unspecified results.

— See the restriction on single subshells described below.

The results of command substitution shall not be processed for further tilde expansion, parameter expansion, command substitution, or arithmetic expansion. If a command substitution occurs inside double-quotes, field splitting and path-name expansion shall not be performed on the results of the substitution.

Command substitution can be nested. To specify nesting within the backquoted version, the application shall precede the inner backquotes with backslashes; for example,

\‘*command* \‘

If the command substitution consists of a single subshell, such as

$((*command*))

a conforming application shall separate the $(and (into two tokens (i.e., separate them with white space).

3.6.3.1 Command Substitution Rationale. *(This subclause is not a part of P1003.2)*

The new $( ) form of command substitution was adopted from the KornShell to solve a problem of inconsistent behavior when using backquotes. For example:

Command	Output
`echo '\$x'`	`\$x`
`echo ‘echo '\$x'‘`	`$x`
`echo $(echo '\$x')`	`\$x`

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the new $( ) form can process any kind of valid embedded script, the backquoted cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

```
echo ‘                          echo $(
cat <<\eof                      cat <<\eof
a here-doc with ‘               a here-doc with )
eof                             eof
‘                               )

echo ‘                          echo $(
echo abc # a comment with ‘     echo abc # a comment with )
‘                               )

echo ‘                          echo $(
echo ’‘’                        echo ')'
‘                               )
```

Some historical KornShell implementations did not process the first two examples correctly, but the author has agreed to make the appropriate modifications to do

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

so. The KornShell will also be modified so that the following works:

```
echo $(
        case word in
                [Ff]oo) echo found foo ;;
        esac
)
```

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts. Because of its widespread historical use, particularly by interactive users, however, the backquotes were retained in POSIX. 2 without being declared obsolescent.

The KornShell feature:

> If *command* is of the form `<`*word* , *word* is expanded to generate a pathname, and the value of the command substitution is the contents of this file with any trailing `<newline>`s deleted.

was omitted from this standard because `$(cat word)` is an appropriate substitute. However, to prevent breaking numerous scripts relying on this feature, it is unspecified to have a script within `$( )` that has only redirections.

The requirement to separate `$(` and `(` when a single subshell is command-substituted is to avoid any ambiguities with Arithmetic Expansion. See 3.6.4.1.

3.6.4 Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion shall be as follows:

`$((`*expression* `))`

The expression shall be treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell shall expand all tokens in the expression for parameter expansion, command substitution, and quote removal.

Next, the shell shall treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression shall be processed according to the rules given in 2.9.2.1, with the following exceptions:

(1) Only integer arithmetic is required.

(2) The `sizeof( )` operator and the prefix and postfix `++` and `--` operators are not required.

(3) Selection, Iteration, and Jump Statements are not supported.

As an extension, the shell may recognize arithmetic expressions beyond those listed. If the expression is invalid, the expansion fails and the shell shall write a message to standard error indicating the failure.

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

944 **3.6.4.1 Arithmetic Expansion Rationale.** *(This subclause is not a part of P1003.2)*

945 Numerous ballots were received objecting to the inclusion of the `(( ))` form of
946 KornShell arithmetic in previous drafts. The developers of the standard con-
947 cluded that there is a strong desire for some kind of arithmetic evaluator to
948 replace `expr`, and that tying it in with `$` makes it fit in nicely with the standard
949 shell language, and provides access to arithmetic evaluation in places where
950 accessing a utility would be inconvenient or clumsy.

951 Following long debate by interested members of the balloting group, the syntax
952 and semantics for arithmetic were changed. The language is essentially a pure
953 arithmetic evaluator of constants and operators (excluding assignment) and
954 represents a simple subset of the previous arithmetic language [which was
955 derived from the KornShell's `(( ))` construct]. The syntax was changed from
956 that of a command denoted by `((`*expression* `))`, to an expansion denoted by
957 `$((`*expression* `))`. The new form is a dollar expansion (`$`), which evaluates the
958 expression and substitutes the resulting value. Objections to the previous style of
959 arithmetic included that it was too complicated, did not fit in well with the shell's
960 use of variables, and the syntax conflicted with subshells. The justification for the
961 new syntax is that the shell is traditionally a macro language, and if a new
962 feature is to be added, it should be done by extending the capabilities presented
963 by the current model of the shell, rather than by inventing a new one outside the
964 model: adding a new dollar expansion was perceived to be the most intuitive and
965 least destructive way to add such a new capability.

966 In Drafts 9 and 10, a form `$[`*expression* `]` was used. It was functionally
967 equivalent to the `$(( ))` of the current text, but objections were lodged that the
968 1988 KornShell had already implemented `$(( ))` and there was no compelling
969 reason to invent yet another syntax. Furthermore, the `$[]` syntax had a minor
970 incompatibility involving the patterns in `case` statements.

971 The portion of the C Standard {7} arithmetic operations selected corresponds to
972 the operations historically supported in the KornShell.

973 A simple example using arithmetic expansion:

```
974     # repeat a command 100 times
975     x=100
976     while [ $x -gt 0 ]
977     do
978             command
979             x=$(($x-1))
980     done
```

981 It was concluded that the `test` command (`[`) was sufficient for the majority of
982 relational arithmetic tests, and that tests involving complicated relational expres-
983 sions within the shell are rare, yet could still be accommodated by testing the
984 value of `$(())` itself. For example:

```
985     # a complicated relational expression
986     while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) -ne 0 ]
```

987 or better yet, the rare script that has many complex relational expressions could

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

define a function like this:

```
val() {
        return $((!$1))
}
```

and complicated tests would be less intimidating:

```
while val $(( (($x + $y)/($a * $b)) < ($foo*$bar) ))
do
        # some calculations
done
```

Another suggestion was to modify `true` and `false` to take an optional argument, and `true` would exit true only if the argument is nonzero, and `false` would exit false only if the argument is nonzero. The suggestion was not favorably received by the balloting group (those contacted were negative about it, all others were silent in their latest ballots).

```
while true $(($x > 5 && $y <= 25))
```

There is a minor portability concern with the new syntax. The example `$((2+2))` could have been intended to mean a command substitution of a utility named `2+2` in a subshell. The developers of POSIX.2 consider this to be obscure and isolated to some KornShell scripts [because `$( )` command substitution existed previously only in the KornShell]. The text on Command Substitution has been changed to require that the `$(` and `(` be separate tokens if this usage is needed.

An example such as

```
echo $((echo hi);(echo there))
```

should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. However, as indicated by 3.1.1, a conforming application must separate two adjacent parentheses with white space to indicate nested subshells.

3.6.5 Field Splitting

After parameter expansion (3.6.2), command substitution (3.6.3), and arithmetic expansion (3.6.4) the shell shall scan the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell shall treat each character of the **IFS** as a delimiter and use the delimiters to split the results of parameter expansion and command substitution into fields.

(1) If the value of **IFS** is `<space>`, `<tab>`, and `<newline>`, or if it is unset, any sequence of `<space>`, `<tab>`, or `<newline>` characters at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall delimit a field. (For example, the input

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

```
<newline><space><tab>foo<tab><tab>bar<space>
```

yields two fields, `foo` and `bar`).

(2) If the value of **IFS** is null, no field splitting shall be performed.

(3) Otherwise, the following rules shall be applied in sequence. The term ‘**IFS** white space’’ is used to mean any sequence (zero or more instances) of white-space characters that are in the **IFS** value (e.g., if **IFS** contains `<space><comma><tab>`, any sequence of `<space>` and `<tab>` characters is considered **IFS** white space).

(a) **IFS** white space shall be ignored at the beginning and end of the input.

(b) Each occurrence in the input of an **IFS** character that is not **IFS** white space, along with any adjacent **IFS** white space, shall delimit a field, as described previously.

(c) Nonzero-length **IFS** white space shall delimit a field.

3.6.5.1 Field Splitting Rationale. *(This subclause is not a part of P1003.2)*

The operation of field splitting using **IFS** as described in earlier drafts was based on the way the KornShell splits words, but is incompatible with other common versions of the shell. However, each has merit, and so a decision was made to allow both. If the **IFS** variable is unset, or is `<space><tab><newline>`, the operation is equivalent to the way the System V shell splits words. Using characters outside the `<space><tab><newline>` set yields the KornShell behavior, where each of the non-`<space><tab><newline>` characters is significant. This behavior, which affords the most flexibility, was taken from the way the original `awk` handled field splitting.

The (3) rule can be summarized as a pseudo ERE:

```
(s*ns*|s+)
```

where `s` is an **IFS** white-space character and `n` is a character in the **IFS** that is not white space. Any string matching that ERE delimits a field, except that the `s+` form does not delimit fields at the beginning or the end of a line. For example, if **IFS** is `<space><comma>`, the string

```
<space><space>red<space><space>,<space>white<space>blue
```

yields the three colors as the delimited fields.

3.6.6 Pathname Expansion

After field splitting, if `set -f` is not in effect, each field in the resulting command line shall be expanded using the algorithm described in 3.13, qualified by the rules in 3.13.3.

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

3.6.7 Quote Removal

The quote characters

```
\ ' "
```

(backslash, single-quote, double-quote) that were present in the original word shall be removed unless they have themselves been quoted.

3.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see 3.12) or for any command. *Redirection operators* can be used with numbers representing file descriptors (see the definition in POSIX.1 {8}) as described below. See also 2.9.1. The relationship between these file descriptors and access to them in a programming language is specified in the language binding for that language to this standard.

The overall format used for redirection is:

```
[n]redir-op word
```

The number *n* is an optional decimal number designating the file descriptor number; it shall be delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number shall not be recognized as part of the redirection expression. (For example, `echo \2>a` writes the character `2` into file `a`). If any part of *redir-op* is quoted, no redirection expression shall be recognized. (For example, `echo 2\>a` writes the characters `2>a` to standard output.) The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

In this standard, open files are represented by decimal numbers starting with zero. It is implementation defined what the largest value can be; however, all implementations shall support at least 0 through 9 for use by the application. These numbers are called *file descriptors*. The values 0, 1, and 2 have special meaning and conventional uses and are implied by certain redirection operations; they are referred to as *standard input*, *standard output*, and *standard error*, respectively. Programs usually take their input from standard input, and write output on standard output. Error messages are usually written to standard error. The redirection operators can be preceded by one or more digits (with no intervening `<blank>`s allowed) to designate the file descriptor number.

If the redirection operator is `<<` or `<<-`, the word that follows the redirection operator shall be subjected to quote removal; it is unspecified whether any of the other expansions occur. For the other redirection operators, the word that follows the redirection operator shall be subjected to tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion shall not be performed on the word by a noninteractive shell; an interactive shell may perform it, but shall do so only when the expansion would result in one word.

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

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

In the following description of redirections, references are made to opening and creating files. These references shall conform to the requirements in 2.9.1.4. A failure to open or create a file shall cause the redirection to fail.

3.7.1 Redirecting Input

Input redirection shall cause the file whose name results from the expansion of *word* to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

[*n*]<*word*

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard input (file descriptor `0`).

3.7.2 Redirecting Output

The two general formats for redirecting output are:

[*n*]>*word*
[*n*]>|*word*

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard output (file descriptor `1`).

Output redirection using the > format shall fail if the *noclobber* option is set (see the description of `set –C` in 3.14.11) and the file named by the expansion of *word* exists and is a regular file. Otherwise, redirection using the > or >| formats shall cause the file whose name results from the expansion of *word* to be created and opened for ouput on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

3.7.3 Appending Redirected Output

Appended output redirection shall cause the file whose name results from the expansion of word to be opened for output on the designated file descriptor. The file is opened as if the POSIX .1 {8} *open* () function was called with the O_APPEND flag. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

[*n*]>>*word*

where the optional *n* represents the file descriptor number.

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

3.7.4 Here-Document

The redirection operators << and <<– both allow redirection of lines contained in a shell input file, known as a *here-document* , to the standard input of a command.

The here-document shall be treated as a single word that begins after the next <newline> and continues until there is a line containing only the delimiter, with no trailing <blank>s. Then the next here-document starts, if there is one. The format is as follows:

[*n*]<<*word*
here-document
delimiter

If any character in *word* is quoted, the delimiter shall be formed by performing quote removal on *word* , and the here-document lines shall not be expanded. Otherwise, the delimiter shall be the *word* itself.

If no characters in *word* are quoted, all lines of the here-document shall be expanded for parameter expansion, command substitution, and arithmetic expansion. In this case, the backslash in the input shall behave as the backslash inside double-quotes (see 3.2.3). However, the double-quote character (") shall not be treated specially within a here-document, except when the double-quote appears within $( ), ` `, or ${ }. 1

If the redirection symbol is <<–, all leading <tab> characters shall be stripped from input lines and the line containing the trailing delimiter. If more than one << or <<– operator is specified on a line, the here-document associated with the first operator shall be supplied first by the application and shall be read first by the shell.

3.7.5 Duplicating an Input File Descriptor

The redirection operator

[*n*]<&*word*

is used to duplicate one input file descriptor from another, or to close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard input if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word* ; if the digits in *word* do not represent a file descriptor already open for input, a redirection error shall result (see 3.8.1). If *word* evaluates to –, file 1
descriptor *n*, or standard input if *n* is not specified, shall be closed. If *word* evaluates to something else, the behavior is unspecified.

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

3.7.6 Duplicating an Output File Descriptor

The redirection operator

[*n***]>&***word*

is used to duplicate one output file descriptor from another, or to close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard output if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for output, a redirection error shall result (see 3.8.1). If *word* evaluates to −, file descriptor *n*, or standard output if *n* is not specified, shall be closed. If *word* evaluates to something else, the behavior is unspecified.

3.7.7 Open File Descriptors for Reading and Writing.

The redirection operator

[*n***]<>***word*

shall cause the file whose name is the expansion of *word* to be opened for both reading and writing on the file descriptor denoted by *n*, or standard input if *n* is not specified. If the file does not exist, it shall be created.

3.7.8 Redirection Rationale. *(This subclause is not a part of P1003.2)*

In the C binding for POSIX.1 {8}, file descriptors are integers in the range 0 − ({OPEN_MAX}−1). The file descriptors discussed in Redirection are that same set of small integers.

As POSIX.2 is being finalized, it is not known how file descriptors will be represented in the language-independent description of POSIX.1 {8}. The current consensus appears to be that they will remain as small integers, but it is still possible that they will be defined as an opaque type. If they remain as integers, then the current POSIX.2 wording is acceptable. If they become an opaque type, then the C binding to POSIX.1 {8} will have to define the mapping from the binding's small integers to the opaque type, and the Redirection clause in POSIX.2 will have to be modified to specify that same mapping.

Having multidigit file descriptor numbers for I/O redirection can cause some obscure compatibility problems. Specifically, scripts that depend on an example command:

```
echo 22>/dev/null
```

echoing `"2"` are somewhat broken to begin with. However, the file descriptor number still must be delimited from the preceding text. For example,

```
cat file2>foo
```

will write the contents of `file2`, not the contents of `file`.

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

The `>|` format of output redirection was adopted from the KornShell. Along with the *noclobber* option, `set –C`, it provides a safety feature to prevent inadvertent overwriting of existing files. (See the rationale with the `pathchk` utility for why this step was taken.) The restriction on regular files is historical practice.

The System V shell and the KornShell have differed historically on pathname expansion of *word* ; the former never performed it, the latter only when the result was a single field (file). As a compromise, it was decided that the KornShell functionality was useful, but only as a shorthand device for interactive users. No reasonable shell script would be written with a command such as:

```
cat foo > a*
```

Thus, shell scripts are prohibited from doing it, while interactive users can select the shell with which they are most comfortable.

The construct `2>&1` is often used to redirect standard error to the same file as standard output. Since the redirections take place beginning to end, the order of redirections is significant. For example:

```
ls > foo 2>&1
```

directs both standard output and standard error to file `foo`. However

```
ls 2>&1 > foo
```

only directs standard output to file `foo` because standard error was duplicated as standard output before standard output was directed to file `foo`.

The `<>` operator is a feature first documented in the KornShell, but it has been silently present in both System V and BSD shells. It could be useful in writing an application that worked with several terminals, and occasionally wanted to start up a shell. That shell would in turn be unable to run applications that run from an ordinary controlling terminal unless it could make use of `<>` redirection. The specific example is a historical version of the pager `more`, which reads from standard error to get its commands, so standard input and standard output are both available for their usual usage. There is no way of saying the following in the shell without `<>`:

```
cat food | more - >/dev/tty03 2<>/dev/tty03
```

Another example of `<>` is one that opens `/dev/tty` on file descriptor 3 for reading and writing:

```
exec 3<> /dev/tty
```

An example of creating a lock file for a critical code region:

```
set -C
until   2> /dev/null > lockfile
do      sleep 30
done
set +C
perform critical function
rm lockfile
```

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

Since `/dev/null` is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

The case of a missing delimiter at the end of a here-document is not specified. This is considered an error in the script (one that sometimes can be difficult to diagnose), although some systems have treated end-of-file as an implicit delimiter.

Tilde expansion is not performed on a here-document because the data is treated as if it were enclosed in double-quotes.

3.8 Exit Status and Errors

3.8.1 Consequences of Shell Errors

For a noninteractive shell, an error condition encountered by a special built-in (see 3.14) or other type of utility shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

	Special Built-in	Other Utilities
Shell language syntax error	shall exit	shall exit
Utility syntax error (option or operand error)	shall exit	shall not exit
Redirection error	shall exit	shall not exit
Variable assignment error	shall exit	shall not exit
Expansion error	shall exit	shall exit
Command not found	n/a	may exit
`dot` script not found	shall exit	n/a

An ‘‘expansion error’’ is one that occurs when the shell expansions defined in 3.6 are carried out (e.g., `${x!y}`, because `!` is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as ‘‘shall (may) exit’’ occur in a subshell, the subshell shall (may) exit with a nonzero status, but the script containing the subshell shall not exit because of the error.

In all of the cases shown in the table, an interactive shell shall write a diagnostic message to standard error without exiting.

3.8.2 Exit Status for Commands

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities are documented in this subclause. The exit status of the standard utilities are documented in their respective clauses.

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

If a command is not found by the shell, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. See 3.9.1.1. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status shall be greater than zero.

Internally, for purposes of deciding if a command exits with a nonzero exit status, the shell shall recognize the entire status value retrieved for the command by the equivalent of the POSIX .1 {8} *wait* () function WEXITSTATUS macro. When reporting the exit status with the special parameter ?, the shell shall report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal shall be reported as greater than 128.

3.8.3 Exit Status and Errors Rationale. *(This subclause is not a part of P1003.2)*

There is a historical difference in `sh` and `ksh` noninteractive error behavior. When a command named in a script is not found, some implementations of `sh` exit immediately, but `ksh` continues with the next command. Thus, POSIX. 2 says that the shell ‘‘may’’ exit in this case. This puts a small burden on the programmer, who will have to test for successful completion following a command if it is important that the next command not be executed if the previous was not found. If it is important for the command to have been found, it was probably also important for it to complete successfully. The test for successful completion would not need to change.

Historically, shells have returned an exit status of 128+ *n*, where *n* represents the signal number. Since signal numbers are not standardized, there is no portable way to determine which signal caused the termination. Also, it is possible for a command to exit with a status in the same range of numbers that the shell would use to report that the command was terminated by a signal. Implementations are encouraged to chose exit values greater than 256 to indicate programs that terminated by a signal so that the exit status cannot be confused with an exit status generated by a normal termination.

Historical shells make the distinction between ‘‘utility not found’’ and ‘‘utility found but cannot execute’’ in their error messages. By specifying two seldomly used exit status values for these cases, 127 and 126 respectively, this gives an application the opportunity to make use of this distinction without having to parse an error message that would probably change from locale to locale. The POSIX. 2 `command`, `env`, `nohup`, and `xargs` utilities also have been specified to use this convention.

When a command fails during word expansion or redirection, most historical implementations exit with a status of 1. However, there was some sentiment that this value should probably be much higher, so that an application could distinguish this case from the more normal exit status values. Thus, the language ‘‘greater than zero’’ was selected to allow either method to be implemented.

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

3.9 Shell Commands

This clause describes the basic structure of shell commands. The following command descriptions each describe a format of the command that is only used to aid the reader in recognizing the command type, and does not formally represent the syntax. Each description discusses the semantics of the command; for a formal description of the command language, consult the grammar in 3.10.

A *command* is one of the following:

— *simple command* (see 3.9.1)

— *pipeline* (see 3.9.2)

— *list* or *compound-list* (see 3.9.3)

— *compound command* (see 3.9.4)

— *function definition* (see 3.9.5).

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command. There is no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, etc.).

3.9.0.1 Shell Commands Rationale. *(This subclause is not a part of P1003.2)*

A description of an ‘‘empty command’’ was removed from an earlier draft because it is only relevant in the cases of `sh -c ""`, `system("")`, or an empty shell-script file (such as the implementation of `true` on some historical systems). Since it is no longer mentioned in POSIX.2, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications will not construct empty commands. (However, note that `sh` does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra `<newline>`s or semicolons, such as

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

3.9.1 Simple Commands

A *simple command* is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (i.e., when any conditional construct such as an AND-OR list or a `case` statement has not bypassed the

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

simple command), the following expansions, assignments, and redirections shall all be performed from the beginning of the command text to the end.

(1) The words that are recognized as variable assignments or redirections according to 3.10.2 are saved for processing in steps (3) and (4).

(2) The words that are not variable assignments or redirections shall be expanded. If any fields remain following their expansion, the first field shall be considered the command name, and remaining fields shall be the arguments for the command.

(3) Redirections shall be performed as described in 3.7.

(4) Each variable assignment shall be expanded for tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal prior to assigning the value.

In the preceding list, the order of steps (3) and (4) may be reversed for the processing of special built-in utilities. See 3.14.

If no command name results, variable assignments shall affect the current execution environment. Otherwise, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment (except for special built-ins). If any of the variable assignments attempt to assign a value to a read-only variable, a variable assignment error shall occur. See 3.8.1 for the consequences of these errors.

If there is no command name, any redirections shall be performed in a subshell environment; it is unspecified whether this subshell environment is the same one as that used for a command substitution within the command. [To affect the current execution environment, see `exec` (3.14.6)]. If any of the redirections performed in the current shell execution environment fail, the command shall immediately fail with an exit status greater than zero, and the shell shall write an error message indicating the failure. See 3.8.1 for the consequences of these failures on interactive and noninteractive shells.

If there is a command name, execution shall continue as described in 3.9.1.1. If there is no command name, but the command contained a command substitution, the command shall complete with the exit status of the last command substitution performed. Otherwise, the command shall complete with a zero exit status.

3.9.1.0.1 Simple Commands Rationale. *(This subclause is not a part of P1003.2)*

The enumerated list is used only when the command is actually going to be executed. For example, in:

```
true || $foo *
```

no expansions are performed.

The following example illustrates both how a variable assignment without a command name affects the current execution environment, and how an assignment with a command name only affects the execution environment of the command.

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

```
$ x=red
$ echo $x
red
$ export x
$ sh –c 'echo $x'
red
$ x=blue sh –c 'echo $x'
blue
$ echo $x
red
```

This next example illustrates that redirections without a command name are still performed.

```
$ ls foo
ls: foo: no such file or directory
$ > foo
$ ls foo
foo
```

Historical practice is for a command without a command name, but that includes a command substitution, to have an exit status of the last command substitution that the shell performed and some historical scripts rely on this. For example:

```
if      x=$(command )
then    ...
fi
```

An example of redirections without a command name being performed in a subshell shows that the here-document does not disrupt the standard input of the `while` loop:

```
IFS=:
while   read a b
do      echo $a
        <<-eof
        Hello
        eof
done </etc/passwd
```

Some examples of commands without command names in AND/OR lists:

```
> foo || {
        echo "error: foo cannot be created" >&2
        exit 1
}

# set saved if /vmunix.save exists
test -f /vmunix.save && saved=1
```

Command substitution and redirections without command names both occur in subshells, but they are not the same ones. For example, in:

```
exec 3> file
var=$(echo foo >&3) 3>&1
```

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

it is unspecified whether `foo` will be echoed to the file or to standard output.

3.9.1.1 Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions shall be performed.

(1) If the command name does not contain any slashes, the first successful step in the following sequence shall occur:

(a) If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.

(b) If the command name matches the name of a function known to this shell, the function shall be invoked as described in 3.9.5. [If the implementation has provided a standard utility in the form of a function, it shall not be recognized at this point. It shall be invoked in conjunction with the path search in step (1)(d).]

(c) If the command name matches the name of a utility listed in Table 2-2 (see 2.3), that utility shall be invoked.

(d) Otherwise, the command shall be searched for using the **PATH** environment variable as described in 2.6:

[1] If the search is successful:

[a] If the system has implemented the utility as a regular built-in or as a shell function, it shall be invoked at this point in the path search.

[b] Otherwise, the shell shall execute the utility in a separate utility environment (see 3.12) with actions equivalent to calling the POSIX.1 {8} *execve*() function with the *path* argument set to the pathname resulting from the search, *arg0* set to the command name, and the remaining arguments set to the operands, if any.

If the *execve*() function fails due to an error equivalent to the POSIX.1 {8} error [ENOEXEC], the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message, and return an exit status of 126.

Once a utility has been searched for and found (either as a result of this specific search or as part of an unspecified shell startup activity), an implementation may remember its location and need not search for the utility again unless the **PATH** variable has been the subject of an assignment. If the remembered location fails for a subsequent invocation, the shell shall

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

repeat the search to find the new location for the utility, if any.

[2] If the search is unsuccessful, the command shall fail with an exit status of 127 and the shell shall write an error message.

(2) If the command name does contain slashes, the shell shall execute the utility in a separate utility environment with actions equivalent to calling the POSIX.1 {8} *execve*() function with the *path* and *arg0* arguments set to the command name, and the remaining arguments set to the operands, if any.

If the *execve*() function fails due to an error equivalent to the POSIX.1 {8} error [ENOEXEC], the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message, and return an exit status of 126.

3.9.1.1.1 Command Search and Execution Rationale. *(This subclause is not a part of P1003.2)*

This description requires that the shell can execute shell scripts directly, even if the underlying system does not support the common #! interpreter convention. That is, if file `foo` contains shell commands and is executable, the following will execute `foo`:

```
./foo
```

The command search shown here does not match all historical implementations. A more typical sequence has been:

— Any built-in, special or regular.

— Functions.

— Path search for executable files.

But there are problems with this sequence. Since the programmer has no idea in advance which utilities might have been built into the shell, a function cannot be used to portably override a utility of the same name. (For example, a function named `cd` cannot be written for many historical systems.) Furthermore, the **PATH** variable is partially ineffective in this case and only a pathname with a slash can be used to ensure a specific executable file is invoked.

The sequence selected for POSIX.2 acknowledges that special built-ins cannot be overridden, but gives the programmer full control over which versions of other utilities are executed. It provides a means of suppressing function lookup (via the `command` utility; see 4.12) for the user's own functions and ensures that any regular built-ins or functions provided by the implementation are under the control of the path search. The mechanisms for associating built-ins or functions with executable files in the path are not specified by POSIX.2, but the wording requires that if either is implemented, the application will not be able to distinguish a function or built-in from an executable (other than in terms of performance,

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