NAME

bash − GNU Bourne-Again SHell

SYNOPSIS

bash [options] [command_string | file]

COPYRIGHT

DESCRIPTION

Bash is a command language interpreter that executes commands read from the standard input, from a string, or from a file. It is a reimplementation and extension of the Bourne shell, the historical Unix command language interpreter. **Bash** also incorporates useful features from the *Korn* and *C* shells (**ksh** and **csh**).

POSIX is the name for a family of computing standards based on Unix. **Bash** is intended to be a conformant implementation of the Shell and Utilities portion of the IEEE POSIX specification (IEEE Standard 1003.1). **Bash** POSIX mode (hereafter referred to as *posix mode*) changes the shell's behavior where its default operation differs from the standard to strictly conform to the standard. See **SEE ALSO** below for a reference to a document that details how posix mode affects **bash**'s behavior. **Bash** can be configured to be POSIX-conformant by default.

OPTIONS

All of the single-character shell options documented in the description of the **set** builtin command, including **−o**, can be used as options when the shell is invoked. In addition, **bash** interprets the following options when it is invoked:

−c If the **−c** option is present, then commands are read from the first non-option argument *command_string*. If there are arguments after the *command_string*, the first argument is assigned to **$0** and any remaining arguments are assigned to the positional parameters. The assignment to **$0** sets the name of the shell, which is used in warning and error messages.

−i If the **−i** option is present, the shell is *interactive*.

−l Make **bash** act as if it had been invoked as a login shell (see **INVOCATION** below).

−r If the **−r** option is present, the shell becomes *restricted* (see **RESTRICTED SHELL** below).

−s If the **−s** option is present, or if no arguments remain after option processing, the shell reads commands from the standard input. This option allows the positional parameters to be set when invoking an interactive shell or when reading input through a pipe.

−D Print a list of all double-quoted strings preceded by **$** on the standard output. These are the strings that are subject to language translation when the current locale is not **C** or **POSIX**. This implies the **−n** option; no commands will be executed.

[−+]O [*shopt_option***]**
shopt_option is one of the shell options accepted by the **shopt** builtin (see **SHELL BUILTIN COMMANDS** below). If *shopt_option* is present, **−O** sets the value of that option; **+O** unsets it. If *shopt_option* is not supplied, **bash** prints the names and values of the shell options accepted by **shopt** on the standard output. If the invocation option is **+O**, the output is displayed in a format that may be reused as input.

−− A **−−** signals the end of options and disables further option processing. Any arguments after the **−−** are treated as a shell script filename (see below) and arguments passed to that script. An argument of **−** is equivalent to **−−**.

Bash also interprets a number of multi-character options. These options must appear on the command line before the single-character options to be recognized.

−−debugger
Arrange for the debugger profile to be executed before the shell starts. Turns on extended debugging mode (see the description of the **extdebug** option to the **shopt** builtin below).

−−dump−po−strings
Equivalent to **−D**, but the output is in the GNU *gettext* “po” (portable object) file format.

−−dump−strings
Equivalent to **−D**.

−−help Display a usage message on standard output and exit successfully.

−−init−file *file*
−−rcfile *file*
Execute commands from *file* instead of the standard personal initialization file *~/.bashrc* if the shell is interactive (see **INVOCATION** below).

−−login
Equivalent to **−l**.

−−noediting
Do not use the GNU **readline** library to read command lines when the shell is interactive.

−−noprofile
Do not read either the system-wide startup file */etc/profile* or any of the personal initialization files *~/.bash_profile*, *~/.bash_login*, or *~/.profile*. By default, **bash** reads these files when it is invoked as a login shell (see **INVOCATION** below).

−−norc Do not read and execute the personal initialization file *~/.bashrc* if the shell is interactive. This option is on by default if the shell is invoked as **sh**.

−−posix
Enable posix mode; change the behavior of **bash** where the default operation differs from the POSIX standard to match the standard.

−−restricted
The shell becomes restricted (see **RESTRICTED SHELL** below).

−−verbose
Equivalent to **−v**.

−−version
Show version information for this instance of **bash** on the standard output and exit successfully.

ARGUMENTS

If arguments remain after option processing, and neither the **−c** nor the **−s** option has been supplied, the first argument is treated as the name of a file containing shell commands (a *shell script*). When **bash** is invoked in this fashion, **$0** is set to the name of the file, and the positional parameters are set to the remaining arguments. **Bash** reads and executes commands from this file, then exits. **Bash**’s exit status is the exit status of the last command executed in the script. If no commands are executed, the exit status is 0. **Bash** first attempts to open the file in the current directory, and, if no file is found, searches the directories in **PATH** for the script.

INVOCATION

A *login shell* is one whose first character of argument zero is a **−**, or one started with the **−−login** option.

An *interactive shell* is one started without non-option arguments (unless **−s** is specified) and without the **−c** option, and whose standard input and standard error are both connected to terminals (as determined by *isatty*(3)), or one started with the **−i** option. **Bash** sets **PS1** and **$−** includes **i** if the shell is interactive, so a shell script or a startup file can test this state.

The following paragraphs describe how **bash** executes its startup files. If any of the files exist but cannot be read, **bash** reports an error. Tildes are expanded in filenames as described below under **Tilde Expansion** in the **EXPANSION** section.

When **bash** is invoked as an interactive login shell, or as a non-interactive shell with the **−−login** option, it first reads and executes commands from the file */etc/profile*, if that file exists. After reading that file, it looks for *~/.bash_profile*, *~/.bash_login*, and *~/.profile*, in that order, and reads and executes commands from the first one that exists and is readable. The **--noprofile** option may be used when the shell is started to inhibit this behavior.

When an interactive login shell exits, or a non-interactive login shell executes the **exit** builtin command, **bash** reads and executes commands from the file *~/.bash_logout*, if it exists.

When an interactive shell that is not a login shell is started, **bash** reads and executes commands from *~/.bashrc*, if that file exists. The **--norc** option inhibits this behavior. The **--rcfile** *file* option causes **bash** to use *file* instead of *~/.bashrc*.

When **bash** is started non-interactively, to run a shell script, for example, it looks for the variable **BASH_ENV** in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read and execute. **Bash** behaves as if the following command were executed:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

but does not use the value of the **PATH** variable to search for the filename.

If **bash** is invoked with the name **sh**, it tries to mimic the startup behavior of historical versions of **sh** as closely as possible, while conforming to the POSIX standard as well. When invoked as an interactive login shell, or a non-interactive shell with the **--login** option, it first attempts to read and execute commands from */etc/profile* and *~/.profile*, in that order. The **--noprofile** option inhibits this behavior. When invoked as an interactive shell with the name **sh**, **bash** looks for the variable **ENV**, expands its value if it is defined, and uses the expanded value as the name of a file to read and execute. Since a shell invoked as **sh** does not attempt to read and execute commands from any other startup files, the **--rcfile** option has no effect. A non-interactive shell invoked with the name **sh** does not attempt to read any other startup files.

When invoked as **sh**, **bash** enters posix mode after reading the startup files.

When **bash** is started in posix mode, as with the **--posix** command line option, it follows the POSIX standard for startup files. In this mode, interactive shells expand the **ENV** variable and read and execute commands from the file whose name is the expanded value. No other startup files are read.

Bash attempts to determine when it is being run with its standard input connected to a network connection, as when executed by the historical and rarely-seen remote shell daemon, usually *rshd*, or the secure shell daemon *sshd*. If **bash** determines it is being run non-interactively in this fashion, it reads and executes commands from *~/.bashrc*, if that file exists and is readable. **Bash** does not read this file if invoked as **sh**. The **--norc** option inhibits this behavior, and the **--rcfile** option makes **bash** use a different file instead of *~/.bashrc*, but neither *rshd* nor *sshd* generally invoke the shell with those options or allow them to be specified.

If the shell is started with the effective user (group) id not equal to the real user (group) id, and the **-p** option is not supplied, no startup files are read, shell functions are not inherited from the environment, the **SHELLOPTS**, **BASHOPTS**, **CDPATH**, and **GLOBIGNORE** variables, if they appear in the environment, are ignored, and the effective user id is set to the real user id. If the **-p** option is supplied at invocation, the startup behavior is the same, but the effective user id is not reset.

DEFINITIONS

The following definitions are used throughout the rest of this document.

blank A space or tab.

whitespace
: A character belonging to the **space** character class in the current locale, or for which *isspace*(3) returns true.

word A sequence of characters considered as a single unit by the shell. Also known as a **token**.

name A *word* consisting only of alphanumeric characters and underscores, and beginning with an alphabetic character or an underscore. Also referred to as an **identifier**.

metacharacter
: A character that, when unquoted, separates words. One of the following:
| & ; ( ) < > space tab newline

control operator

A *token* that performs a control function. It is one of the following symbols:
|| & && ; ;; ;& ;;& ( ) | |& <newline>

RESERVED WORDS

Reserved words are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a command (see **SHELL GRAMMAR** below), the third word of a **case** or **select** command (only **in** is valid), or the third word of a **for** command (only **in** and **do** are valid):

! case coproc do done elif else esac fi for function if in select then until while { } time [[ ]]

SHELL GRAMMAR

This section describes the syntax of the various forms of shell commands.

Simple Commands

A *simple command* is a sequence of optional variable assignments followed by **blank**-separated words and redirections, and terminated by a *control operator*. The first word specifies the command to be executed, and is passed as argument zero. The remaining words are passed as arguments to the invoked command.

The return value of a *simple command* is its exit status, or 128+*n* if the command is terminated by signal *n*.

Pipelines

A *pipeline* is a sequence of one or more commands separated by one of the control operators **|** or **|&**. The format for a pipeline is:

[**time** [**−p**]] [!] *command1* [[**|**|**|&**] *command2* ...]

The standard output of *command1* is connected via a pipe to the standard input of *command2*. This connection is performed before any redirections specified by the *command1*(see **REDIRECTION** below). If **|&** is the pipeline operator, *command1*'s standard error, in addition to its standard output, is connected to *command2*'s standard input through the pipe; it is shorthand for **2>&1 |**. This implicit redirection of the standard error to the standard output is performed after any redirections specified by *command1*.

The return status of a pipeline is the exit status of the last command, unless the **pipefail** option is enabled. If **pipefail** is enabled, the pipeline's return status is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands exit successfully. If the reserved word **!** precedes a pipeline, the exit status of that pipeline is the logical negation of the exit status as described above. If a pipeline is executed synchronously, the shell waits for all commands in the pipeline to terminate before returning a value.

If the **time** reserved word precedes a pipeline, the shell reports the elapsed as well as user and system time consumed by its execution when the pipeline terminates. The **−p** option changes the output format to that specified by POSIX. When the shell is in posix mode, it does not recognize **time** as a reserved word if the next token begins with a "−". The value of the **TIMEFORMAT** variable is a format string that specifies how the timing information should be displayed; see the description of **TIMEFORMAT** below under **Shell Variables**.

When the shell is in posix mode, **time** may appear by itself as the only word in a simple command. In this case, the shell displays the total user and system time consumed by the shell and its children. The **TIMEFORMAT** variable specifies the format of the time information.

Each command in a multi-command pipeline, where pipes are created, is executed in a *subshell*, which is a separate process. See **COMMAND EXECUTION ENVIRONMENT** for a description of subshells and a subshell environment. If the **lastpipe** option is enabled using the **shopt** builtin (see the description of **shopt** below), and job control is not active, the last element of a pipeline may be run by the shell process.

Lists

A *list* is a sequence of one or more pipelines separated by one of the operators **;**, **&**, **&&**, or **||**, and optionally terminated by one of **;**, **&**, or **<newline>**.

Of these list operators, **&&** and **||** have equal precedence, followed by **;** and **&**, which have equal precedence.

A sequence of one or more newlines may appear in a *list* instead of a semicolon to delimit commands.

If a command is terminated by the control operator **&**, the shell executes the command in the *background* in a subshell. The shell does not wait for the command to finish, and the return status is 0. These are referred to as *asynchronous* commands. Commands separated by a **;** are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

AND and OR lists are sequences of one or more pipelines separated by the **&&** and **||** control operators, respectively. AND and OR lists are executed with left associativity. An AND list has the form

command1 **&&** *command2*

command2 is executed if, and only if, *command1* returns an exit status of zero (success).

An OR list has the form

command1 **||** *command2*

command2 is executed if, and only if, *command1* returns a non-zero exit status. The return status of AND and OR lists is the exit status of the last command executed in the list.

Compound Commands

A *compound command* is one of the following. In most cases a *list* in a command's description may be separated from the rest of the command by one or more newlines, and may be followed by a newline in place of a semicolon.

(*list*) *list* is executed in a subshell (see **COMMAND EXECUTION ENVIRONMENT** below for a description of a subshell environment). Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes. The return status is the exit status of *list*.

{ *list*; } *list* is executed in the current shell environment. *list* must be terminated with a newline or semicolon. This is known as a *group command*. The return status is the exit status of *list*.

Note that unlike the metacharacters **(** and **)**, **{** and **}** are *reserved words* and must occur where a reserved word is permitted to be recognized. Since they do not cause a word break, they must be separated from *list* by whitespace or another shell metacharacter.

((*expression*))

The arithmetic *expression* is evaluated according to the rules described below under **ARITHMETIC EVALUATION**. If the value of the expression is non-zero, the return status is 0; otherwise the return status is 1. The *expression* undergoes the same expansions as if it were within double quotes, but unescaped double quote characters in *expression* are not treated specially and are removed. Since this can potentially result in empty strings, this command treats those as expressions that evaluate to 0.

[[*expression* **]]**

Evaluate the conditional expression *expression* and return a status of zero (true) or non-zero (false). Expressions are composed of the primaries described below under **CONDITIONAL EXPRESSIONS**. The words between the **[[** and **]]** do not undergo word splitting and pathname expansion. The shell performs tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal on those words. Conditional operators such as **–f** must be unquoted to be recognized as primaries.

When used with **[[**, the **<** and **>** operators sort lexicographically using the current locale.

When the **==** and **!=** operators are used, the string to the right of the operator is considered a pattern and matched according to the rules described below under **Pattern Matching**, as if the **extglob** shell option were enabled. The **=** operator is equivalent to **==**. If the **nocasematch** shell option is enabled, the match is performed without regard to the case of alphabetic characters. The return value is 0 if the string matches (**==**) or does not match (**!=**) the pattern, and 1 otherwise. If any part of the pattern is quoted, the quoted portion is matched as a string: every character in the quoted portion matches itself, instead of having any special pattern matching meaning.

An additional binary operator, **=~**, is available, with the same precedence as **==** and **!=**. When it is used, the string to the right of the operator is considered a POSIX extended regular expression and matched accordingly (using the POSIX *regcomp* and *regexec* interfaces usually described in *regex*(3)). The return value is 0 if the string matches the pattern, and 1 otherwise. If the regular expression is syntactically incorrect, the conditional expression's return value is 2. If the **nocasematch** shell option is enabled, the match is performed without regard to the case of alphabetic characters.

If any part of the pattern is quoted, the quoted portion is matched literally, as above. If the pattern is stored in a shell variable, quoting the variable expansion forces the entire pattern to be matched literally. Treat bracket expressions in regular expressions carefully, since normal quoting and pattern characters lose their meanings between brackets.

The match succeeds if the pattern matches any part of the string. Anchor the pattern using the **^** and **$** regular expression operators to force it to match the entire string.

The array variable **BASH_REMATCH** records which parts of the string matched the pattern. The element of **BASH_REMATCH** with index 0 contains the portion of the string matching the entire regular expression. Substrings matched by parenthesized subexpressions within the regular expression are saved in the remaining **BASH_REMATCH** indices. The element of **BASH_REMATCH** with index *n* is the portion of the string matching the *n*th parenthesized subexpression. **Bash** sets **BASH_REMATCH** in the global scope; declaring it as a local variable will lead to unexpected results.

Expressions may be combined using the following operators, listed in decreasing order of precedence:

(*expression* **)**
: Returns the value of *expression*. This may be used to override the normal precedence of operators.

! *expression*
: True if *expression* is false.

expression1 **&&** *expression2*
: True if both *expression1* and *expression2* are true.

expression1 **||** *expression2*
: True if either *expression1* or *expression2* is true.

The **&&** and **||** operators do not evaluate *expression2* if the value of *expression1* is sufficient to determine the return value of the entire conditional expression.

for *name* [[**in** *word* ...] ;] **do** *list* ; **done**
: First, expand The list of words following **in**, generating a list of items. Then, the variable *name* is set to each element of this list in turn, and *list* is executed each time. If the **in** *word* is omitted, the **for** command executes *list* once for each positional parameter that is set (see **PARAMETERS** below). The return status is the exit status of the last command that executes. If the expansion of the items following **in** results in an empty list, no commands are executed, and the return status is 0.

for ((*expr1* ; *expr2* ; *expr3*)) [;] **do** *list* ; **done**
: First, evaluate the arithmetic expression *expr1* according to the rules described below under **ARITHMETIC EVALUATION**. Then, repeatedly evaluate the arithmetic expression *expr2* until it evaluates to zero. Each time *expr2* evaluates to a non-zero value, execute *list* and evaluate the arithmetic expression *expr3*. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in *list* that is executed, or non-zero if any of the expressions is invalid.

  Use the **break** and **continue** builtins (see **SHELL BUILTIN COMMANDS** below) to control loop execution.

select *name* [**in** *word*] ; **do** *list* ; **done**

First, expand the list of words following **in**, generating a list of items, and print the set of expanded words the standard error, each preceded by a number. If the **in** *word* is omitted, print the positional parameters (see **PARAMETERS** below). **select** then displays the **PS3** prompt and reads a line from the standard input. If the line consists of a number corresponding to one of the displayed words, then **select** sets the value of *name* to that word. If the line is empty, **select** displays the words and prompt again. If EOF is read, **select** completes and returns 1. Any other value sets *name* to null. The line read is saved in the variable **REPLY**. The *list* is executed after each selection until a **break** command is executed. The exit status of **select** is the exit status of the last command executed in *list*, or zero if no commands were executed.

case *word* **in** [[(] *pattern* [**|** *pattern*] ...) *list* ;;] ... **esac**

A **case** command first expands *word*, and tries to match it against each *pattern* in turn, proceeding from first to last, using the matching rules described under **Pattern Matching** below. A pattern list is a set of one or more patterns separated by , and the) operator terminates the pattern list. The *word* is expanded using tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution and quote removal. Each *pattern* examined is expanded using tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal. If the **nocasematch** shell option is enabled, the match is performed without regard to the case of alphabetic characters. A *clause* is a pattern list and an associated *list*.

When a match is found, **case** executes the corresponding *list*. If the **;;** operator terminates the case clause, the **case** command completes after the first match. Using **;&** in place of **;;** causes execution to continue with the *list* associated with the next pattern list. Using **;;&** in place of **;;** causes the shell to test the next pattern list in the statement, if any, and execute any associated *list* if the match succeeds, continuing the case statement execution as if the pattern list had not matched. The exit status is zero if no pattern matches.

Otherwise, it is the exit status of the last command executed in the last *list* executed.

if *list*; **then** *list*; [**elif** *list*; **then** *list*;] ... [**else** *list*;] **fi**

The **if** *list* is executed. If its exit status is zero, the **then** *list* is executed. Otherwise, each **elif** *list* is executed in turn, and if its exit status is zero, the corresponding **then** *list* is executed and the command completes. Otherwise, the **else** *list* is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.

while *list-1*; **do** *list-2*; **done**
until *list-1*; **do** *list-2*; **done**

The **while** command continuously executes the list *list-2* as long as the last command in the list *list-1* returns an exit status of zero. The **until** command is identical to the **while** command, except that the test is negated: *list-2* is executed as long as the last command in *list-1* returns a non-zero exit status. The exit status of the **while** and **until** commands is the exit status of the last command executed in *list-2*, or zero if none was executed.

Coprocesses

A *coprocess* is a shell command preceded by the **coproc** reserved word. A coprocess is executed asynchronously in a subshell, as if the command had been terminated with the **&** control operator, with a two-way pipe established between the executing shell and the coprocess.

The syntax for a coprocess is:

coproc [*NAME*] *command* [*redirections*]

This creates a coprocess named *NAME*. *command* may be either a simple command or a compound command (see above). *NAME* is a shell variable name. If *NAME* is not supplied, the default name is **COPROC**.

The recommended form to use for a coprocess is

coproc *NAME* { *command* [*redirections*]; }

This form is preferred because simple commands result in the coprocess always being named **COPROC**, and it is simpler to use and more complete than the other compound commands.

If *command* is a compound command, *NAME* is optional. The word following **coproc** determines whether that word is interpreted as a variable name: it is interpreted as *NAME* if it is not a reserved word that introduces a compound command. If *command* is a simple command, *NAME* is not allowed; this is to avoid confusion between *NAME* and the first word of the simple command.

When the coprocess is executed, the shell creates an array variable (see **Arrays** below) named *NAME* in the context of the executing shell. The standard output of *command* is connected via a pipe to a file descriptor in the executing shell, and that file descriptor is assigned to *NAME*[0]. The standard input of *command* is connected via a pipe to a file descriptor in the executing shell, and that file descriptor is assigned to *NAME*[1]. This pipe is established before any redirections specified by the command (see **REDIRECTION** below). The file descriptors can be utilized as arguments to shell commands and redirections using standard word expansions. Other than those created to execute command and process substitutions, the file descriptors are not available in subshells.

The process ID of the shell spawned to execute the coprocess is available as the value of the variable *NAME*_PID. The **wait** builtin may be used to wait for the coprocess to terminate.

Since the coprocess is created as an asynchronous command, the **coproc** command always returns success. The return status of a coprocess is the exit status of *command*.

Shell Function Definitions

A shell function is an object that is called like a simple command and executes a compound command with a new set of positional parameters. Shell functions are declared as follows:

fname () *compound−command* [*redirection*]
function *fname* [()] *compound−command* [*redirection*]

This defines a function named *fname*. The reserved word **function** is optional. If the **function** reserved word is supplied, the parentheses are optional. The *body* of the function is the compound command *compound−command* (see **Compound Commands** above). That command is usually a *list* of commands between { and }, but may be any command listed under **Compound Commands** above. If the **function** reserved word is used, but the parentheses are not supplied, the braces are recommended. *compound−command* is executed whenever *fname* is specified as the name of a simple command. When in posix mode, *fname* must be a valid shell *name* and may not be the name of one of the POSIX *special builtins*. In default mode, a function name can be any unquoted shell word that does not contain **$**.

Any redirections (see **REDIRECTION** below) specified when a function is defined are performed when the function is executed.

The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body. (See **FUNCTIONS** below.)

COMMENTS

In a non-interactive shell, or an interactive shell in which the **interactive_comments** option to the **shopt** builtin is enabled (see **SHELL BUILTIN COMMANDS** below), a word beginning with **#** introduces a comment. A word begins at the beginning of a line, after unquoted whitespace, or after an operator. The comment causes that word and all remaining characters on that line to be ignored. An interactive shell without the **interactive_comments** option enabled does not allow comments. The **interactive_comments** option is enabled by default in interactive shells.

QUOTING

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the *metacharacters* listed above under **DEFINITIONS** has special meaning to the shell and must be quoted if it is to represent itself.

When the command history expansion facilities are being used (see **HISTORY EXPANSION** below), the *history expansion* character, usually **!**, must be quoted to prevent history expansion.

There are four quoting mechanisms: the *escape character*, single quotes, double quotes, and dollar-single quotes.

A non-quoted backslash (\) is the *escape character*. It preserves the literal value of the next character that follows, removing any special meaning it has, with the exception of <newline>. If a ****<newline> pair appears, and the backslash is not itself quoted, the ****<newline> is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of **$**, **`**, ****, and, when history expansion is enabled, **!**. When the shell is in posix mode, the **!** has no special meaning within double quotes, even when history expansion is enabled. The characters **$** and **`** retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: **$**, **`**, **"**, ****, or **<newline>**. Backslashes preceding characters without a special meaning are left unmodified.

A double quote may be quoted within double quotes by preceding it with a backslash. If enabled, history expansion will be performed unless an **!** appearing in double quotes is escaped using a backslash. The backslash preceding the **!** is not removed.

The special parameters ***** and **@** have special meaning when in double quotes (see **PARAMETERS** below).

Character sequences of the form **$**'*string*' are treated as a special variant of single quotes. The sequence expands to *string*, with backslash-escaped characters in *string* replaced as specified by the ANSI C standard. Backslash escape sequences, if present, are decoded as follows:

- **\a** alert (bell)
- **\b** backspace
- **\e**
- **\E** an escape character
- **\f** form feed
- **\n** new line
- **\r** carriage return
- **\t** horizontal tab
- **\v** vertical tab
- **** backslash
- **\'** single quote
- **\"** double quote
- **\?** question mark
- *****nnn* The eight-bit character whose value is the octal value *nnn* (one to three octal digits).
- **\x***HH* The eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits).
- **\u***HHHH*
  The Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHH* (one to four hex digits).
- **\U***HHHHHHHH*
  The Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHHHHHH* (one to eight hex digits).
- **\c***x* A control-*x* character.

The expanded result is single-quoted, as if the dollar sign had not been present.

Translating Strings

A double-quoted string preceded by a dollar sign (**$**"*string*") causes the string to be translated according to the current locale. The *gettext* infrastructure performs the lookup and translation, using the **LC_MESSAGES**, **TEXTDOMAINDIR**, and **TEXTDOMAIN** shell variables. If the current locale is **C** or **POSIX**, if there are no translations available, or if the string is not translated, the dollar sign is ignored, and the string is treated as double-quoted as described above. This is a form of double quoting, so the string remains double-quoted by default, whether or not it is translated and replaced. If the **noexpand_translation** option is enabled using the **shopt** builtin, translated strings are single-quoted instead of double-quoted. See the description of **shopt** below under **SHELL BUILTIN COMMANDS**.

PARAMETERS

A *parameter* is an entity that stores values. It can be a *name*, a number, or one of the special characters listed below under **Special Parameters**. A *variable* is a parameter denoted by a *name*. A variable has a *value* and zero or more *attributes*. Attributes are assigned using the **declare** builtin command (see **declare** below in **SHELL BUILTIN COMMANDS**). The **export** and **readonly** builtins assign specific attributes.

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the **unset** builtin command (see **SHELL BUILTIN COMMANDS** below).

A *variable* is assigned to using a statement of the form

name=[*value*]

If *value* is not given, the variable is assigned the null string. All *values* undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (see **EXPANSION** below). If the variable has its **integer** attribute set, then *value* is evaluated as an arithmetic expression even if the **$((...))** expansion is not used (see **Arithmetic Expansion** below). Word splitting and pathname expansion are not performed. Assignment statements may also appear as arguments to the **alias**, **declare**, **typeset**, **export**, **readonly**, and **local** builtin commands (*declaration* commands). When in posix mode, these builtins may appear in a command after one or more instances of the **command** builtin and retain these assignment statement properties.

In the context where an assignment statement is assigning a value to a shell variable or array index, the "+=" operator appends to or adds to the variable's previous value. This includes arguments to *declaration* commands such as **declare** that accept assignment statements. When "+=" is applied to a variable for which the **integer** attribute has been set, the variable's current value and *value* are each evaluated as arithmetic expressions, and the sum of the results is assigned as the variable's value. The current value is usually an integer constant, but may be an expression. When "+=" is applied to an array variable using compound assignment (see **Arrays** below), the variable's value is not unset (as it is when using and new values are appended to the array beginning at one greater than the array's maximum index (for indexed arrays) or added as additional key−value pairs in an associative array. When applied to a string-valued variable, *value* is expanded and appended to the variable's value.

A variable can be assigned the *nameref* attribute using the **−n** option to the **declare** or **local** builtin commands (see the descriptions of **declare** and **local** below) to create a *nameref*, or a reference to another variable. This allows variables to be manipulated indirectly. Whenever the nameref variable is referenced, assigned to, unset, or has its attributes modified (other than using or changing the *nameref* attribute itself), the operation is actually performed on the variable specified by the nameref variable's value. A nameref is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function. For instance, if a variable name is passed to a shell function as its first argument, running

```
declare −n ref=$1
```

inside the function creates a local nameref variable **ref** whose value is the variable name passed as the first argument. References and assignments to **ref**, and changes to its attributes, are treated as references, assignments, and attribute modifications to the variable whose name was passed as **$1**. If the control variable in a **for** loop has the nameref attribute, the list of words can be a list of shell variables, and a name reference is established for each word in the list, in turn, when the loop is executed. Array variables cannot be given the **nameref** attribute. However, nameref variables can reference array variables and subscripted array variables. Namerefs can be unset using the **−n** option to the **unset** builtin. Otherwise, if **unset** is executed with the name of a nameref variable as an argument, the variable referenced by the nameref variable is unset.

When the shell starts, it reads its environment and creates a shell variable from each environment variable that has a valid name, as described below (see **ENVIRONMENT**).

Positional Parameters

A *positional parameter* is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the **set** builtin command. Positional parameters may not be assigned to with assignment statements. The positional parameters are temporarily replaced when a shell function is executed (see **FUNCTIONS** below).

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces (see **EXPANSION** below). Without braces, a digit following $ can only refer to one of the first nine positional parameters (**$1−$9**) or the special parameter **$0** (see the next section).

Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed. Special parameters are denoted by one of the following characters.

***** (**$***) Expands to the positional parameters, starting from one. When the expansion is not within double quotes, each positional parameter expands to a separate word. In contexts where word expansions are performed, those words are subject to further word splitting and pathname expansion. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the **IFS** variable. That is, **"$*"** is equivalent to **"$1*c*$2*c*..."**, where *c* is the first character of the value of the **IFS** variable. If **IFS** is unset, the parameters are separated by spaces. If **IFS** is null, the parameters are joined without intervening separators.

@ (**$@**) Expands to the positional parameters, starting from one. In contexts where word splitting is performed, this expands each positional parameter to a separate word; if not within double quotes, these words are subject to word splitting. In contexts where word splitting is not performed, such as the value portion of an assignment statement, this expands to a single word with each positional parameter separated by a space. When the expansion occurs within double quotes, and word splitting is performed, each parameter expands to a separate word. That is, **"$@"** is equivalent to **"$1" "$2" ...** If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the expansion of the beginning part of the original word, and the expansion of the last parameter is joined with the expansion of the last part of the original word. When there are no positional parameters, **"$@"** and **$@** expand to nothing (i.e., they are removed).

(**$#**) Expands to the number of positional parameters in decimal.

? (**$?**) Expands to the exit status of the most recently executed command.

− (**$−**) Expands to the current option flags as specified upon invocation, by the **set** builtin command, or those set by the shell itself (such as the **−i** option).

$** (**$$) Expands to the process ID of the shell. In a subshell, it expands to the process ID of the parent shell, not the subshell.

! (**$!**)Expands to the process ID of the job most recently placed into the background, whether executed as an asynchronous command or using the **bg** builtin (see **JOB CONTROL** below).

0 (**$0**) Expands to the name of the shell or shell script. This is set at shell initialization. If **bash** is invoked with a file of commands, **$0** is set to the name of that file. If **bash** is started with the **−c** option, then **$0** is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the filename used to invoke **bash**, as given by argument zero.

Shell Variables

The shell sets following variables:

_ (**$_**, an underscore) This has a number of meanings depending on context. At shell startup, **_** is set to the pathname used to invoke the shell or shell script being executed as passed in the environment or argument list. Subsequently, it expands to the last argument to the previous simple command executed in the foreground, after expansion. It is also set to the full pathname used to invoke each command executed and placed in the environment exported to that command. When checking mail, **$_** expands to the name of the mail file currently being checked.

BASH Expands to the full filename used to invoke this instance of **bash**.

BASHOPTS
A colon-separated list of enabled shell options. Each word in the list is a valid argument for the **−s** option to the **shopt** builtin command (see **SHELL BUILTIN COMMANDS** below). The options appearing in **BASHOPTS** are those reported as *on* by **shopt**. If this variable is in the environment when **bash** starts up, the shell enables each option in the list before reading any startup files. If this variable is exported, child shells will enable each option in the list. This variable is read-only.

BASHPID
Expands to the process ID of the current **bash** process. This differs from **$$** under certain circumstances, such as subshells that do not require **bash** to be re-initialized. Assignments to **BASHPID** have no effect. If **BASHPID** is unset, it loses its special properties, even if it is subsequently reset.

BASH_ALIASES
An associative array variable whose members correspond to the internal list of aliases as maintained by the **alias** builtin. Elements added to this array appear in the alias list; however, unsetting array elements currently does not remove aliases from the alias list. If **BASH_ALIASES** is unset, it loses its special properties, even if it is subsequently reset.

BASH_ARGC
An array variable whose values are the number of parameters in each frame of the current **bash** execution call stack. The number of parameters to the current subroutine (shell function or script executed with **.** or **source**) is at the top of the stack. When a subroutine is executed, the number of parameters passed is pushed onto **BASH_ARGC**. The shell sets **BASH_ARGC** only when in extended debugging mode (see the description of the **extdebug** option to the **shopt** builtin below). Setting **extdebug** after the shell has started to execute a script, or referencing this variable when **extdebug** is not set, may result in inconsistent values. Assignments to **BASH_ARGC** have no effect, and it may not be unset.

BASH_ARGV
An array variable containing all of the parameters in the current **bash** execution call stack. The final parameter of the last subroutine call is at the top of the stack; the first parameter of the initial call is at the bottom. When a subroutine is executed, the shell pushes the supplied parameters onto **BASH_ARGV**. The shell sets **BASH_ARGV** only when in extended debugging mode (see the description of the **extdebug** option to the **shopt** builtin below). Setting **extdebug** after the shell has started to execute a script, or referencing this variable when **extdebug** is not set, may result in inconsistent values. Assignments to **BASH_ARGV** have no effect, and it may not be unset.

BASH_ARGV0
When referenced, this variable expands to the name of the shell or shell script (identical to **$0**; see the description of special parameter 0 above). Assigning a value to **BASH_ARGV0** sets **$0** to the same value. If **BASH_ARGV0** is unset, it loses its special properties, even if it is subsequently reset.

BASH_CMDS
An associative array variable whose members correspond to the internal hash table of commands as maintained by the **hash** builtin. Adding elements to this array makes them appear in the hash table; however, unsetting array elements currently does not remove command names from the hash table. If **BASH_CMDS** is unset, it loses its special properties, even if it is subsequently reset.

BASH_COMMAND
Expands to the command currently being executed or about to be executed, unless the shell is executing a command as the result of a trap, in which case it is the command executing at the time of the trap. If **BASH_COMMAND** is unset, it loses its special properties, even if it is subsequently reset.

BASH_EXECUTION_STRING
The command argument to the **−c** invocation option.

BASH_LINENO
An array variable whose members are the line numbers in source files where each corresponding member of **FUNCNAME** was invoked. **${BASH_LINENO[*$i*]}** is the line number in the source file (**${BASH_SOURCE[*$i+1*]}**) where **${FUNCNAME[*$i*]}** was called (or **${BASH_LINENO[*$i−1*]}** if referenced within another shell function). Use **LINENO** to obtain

the current line number. Assignments to **BASH_LINENO** have no effect, and it may not be unset.

BASH_LOADABLES_PATH
A colon-separated list of directories in which the **enable** command looks for dynamically loadable builtins.

BASH_MONOSECONDS
Each time this variable is referenced, it expands to the value returned by the system's monotonic clock, if one is available. If there is no monotonic clock, this is equivalent to **EPOCHSECONDS**. If **BASH_MONOSECONDS** is unset, it loses its special properties, even if it is subsequently reset.

BASH_REMATCH
An array variable whose members are assigned by the **=~** binary operator to the **[[** conditional command. The element with index 0 is the portion of the string matching the entire regular expression. The element with index *n* is the portion of the string matching the *n*th parenthesized subexpression.

BASH_SOURCE
An array variable whose members are the source filenames where the corresponding shell function names in the **FUNCNAME** array variable are defined. The shell function **${FUNCNAME[***$i***]}** is defined in the file **${BASH_SOURCE[***$i***]}** and called from **${BASH_SOURCE[***$i+1***]}**. Assignments to **BASH_SOURCE** have no effect, and it may not be unset.

BASH_SUBSHELL
Incremented by one within each subshell or subshell environment when the shell begins executing in that environment. The initial value is 0. If **BASH_SUBSHELL** is unset, it loses its special properties, even if it is subsequently reset.

BASH_TRAPSIG
Set to the signal number corresponding to the trap action being executed during its execution. See the description of **trap** under **SHELL BUILTIN COMMANDS** below for information about signal numbers and trap execution.

BASH_VERSINFO
A readonly array variable whose members hold version information for this instance of **bash**. The values assigned to the array members are as follows:

BASH_VERSINFO[0**]**	The major version number (the *release*).
BASH_VERSINFO[1**]**	The minor version number (the *version*).
BASH_VERSINFO[2**]**	The patch level.
BASH_VERSINFO[3**]**	The build version.
BASH_VERSINFO[4**]**	The release status (e.g., *beta*).
BASH_VERSINFO[5**]**	The value of **MACHTYPE**.

BASH_VERSION
Expands to a string describing the version of this instance of **bash** (e.g., 5.2.37(3)-release).

COMP_CWORD
An index into **${COMP_WORDS}** of the word containing the current cursor position. This variable is available only in shell functions invoked by the programmable completion facilities (see **Programmable Completion** below).

COMP_KEY
The key (or final key of a key sequence) used to invoke the current completion function. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).

COMP_LINE
The current command line. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).

COMP_POINT
The index of the current cursor position relative to the beginning of the current command. If the current cursor position is at the end of the current command, the value of this variable is equal to **${#COMP_LINE}**. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).

COMP_TYPE
Set to an integer value corresponding to the type of attempted completion that caused a completion function to be called: *TAB*, for normal completion, *?*, for listing completions after successive tabs, *!*, for listing alternatives on partial word completion, *@*, to list completions if the word is not unmodified, or *%*, for menu completion. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).

COMP_WORDBREAKS
The set of characters that the **readline** library treats as word separators when performing word completion. If **COMP_WORDBREAKS** is unset, it loses its special properties, even if it is subsequently reset.

COMP_WORDS
An array variable (see **Arrays** below) consisting of the individual words in the current command line. The line is split into words as **readline** would split it, using **COMP_WORDBREAKS** as described above. This variable is available only in shell functions invoked by the programmable completion facilities (see **Programmable Completion** below).

COPROC
An array variable (see **Arrays** below) created to hold the file descriptors for output from and input to an unnamed coprocess (see **Coprocesses** above).

DIRSTACK
An array variable (see **Arrays** below) containing the current contents of the directory stack. Directories appear in the stack in the order they are displayed by the **dirs** builtin. Assigning to members of this array variable may be used to modify directories already in the stack, but the **pushd** and **popd** builtins must be used to add and remove directories. Assigning to this variable does not change the current directory. If **DIRSTACK** is unset, it loses its special properties, even if it is subsequently reset.

EPOCHREALTIME
Each time this parameter is referenced, it expands to the number of seconds since the Unix Epoch (see *time*(3)) as a floating-point value with micro-second granularity. Assignments to **EPOCHREALTIME** are ignored. If **EPOCHREALTIME** is unset, it loses its special properties, even if it is subsequently reset.

EPOCHSECONDS
Each time this parameter is referenced, it expands to the number of seconds since the Unix Epoch (see *time*(3)). Assignments to **EPOCHSECONDS** are ignored. If **EPOCHSECONDS** is unset, it loses its special properties, even if it is subsequently reset.

EUID Expands to the effective user ID of the current user, initialized at shell startup. This variable is readonly.

FUNCNAME
An array variable containing the names of all shell functions currently in the execution call stack. The element with index 0 is the name of any currently-executing shell function. The bottom-most element (the one with the highest index) is "main". This variable exists only when a shell function is executing. Assignments to **FUNCNAME** have no effect. If **FUNCNAME** is unset, it loses its special properties, even if it is subsequently reset.

This variable can be used with **BASH_LINENO** and **BASH_SOURCE**. Each element of **FUNCNAME** has corresponding elements in **BASH_LINENO** and **BASH_SOURCE** to describe the call stack. For instance, **${FUNCNAME[*$i*]}** was called from the file **${BASH_SOURCE[*$i+1*]}** at line number **${BASH_LINENO[*$i*]}**. The **caller** builtin displays the current call stack using this information.

GROUPS
An array variable containing the list of groups of which the current user is a member. Assignments to **GROUPS** have no effect. If **GROUPS** is unset, it loses its special properties, even if it is subsequently reset.

HISTCMD
The history number, or index in the history list, of the current command. Assignments to **HISTCMD** have no effect. If **HISTCMD** is unset, it loses its special properties, even if it is subsequently reset.

HOSTNAME
Automatically set to the name of the current host.

HOSTTYPE
Automatically set to a string that uniquely describes the type of machine on which **bash** is executing. The default is system-dependent.

LINENO
Each time this parameter is referenced, the shell substitutes a decimal number representing the current sequential line number (starting with 1) within a script or function. When not in a script or function, the value substituted is not guaranteed to be meaningful. If **LINENO** is unset, it loses its special properties, even if it is subsequently reset.

MACHTYPE
Automatically set to a string that fully describes the system type on which **bash** is executing, in the standard GNU *cpu-company-system* format. The default is system-dependent.

MAPFILE
An array variable (see **Arrays** below) created to hold the text read by the **mapfile** builtin when no variable name is supplied.

OLDPWD
The previous working directory as set by the **cd** command.

OPTARG
The value of the last option argument processed by the **getopts** builtin command (see **SHELL BUILTIN COMMANDS** below).

OPTIND
The index of the next argument to be processed by the **getopts** builtin command (see **SHELL BUILTIN COMMANDS** below).

OSTYPE
Automatically set to a string that describes the operating system on which **bash** is executing. The default is system-dependent.

PIPESTATUS
An array variable (see **Arrays** below) containing a list of exit status values from the commands in the most-recently-executed foreground pipeline, which may consist of only a simple command (see **SHELL GRAMMAR** above). **Bash** sets **PIPESTATUS** after executing multi-element pipelines, timed and negated pipelines, simple commands, subshells created with the (operator, the **[[** and **((** compound commands, and after error conditions that result in the shell aborting command execution.

PPID The process ID of the shell's parent. This variable is readonly.

PWD The current working directory as set by the **cd** command.

RANDOM
Each time this parameter is referenced, it expands to a random integer between 0 and 32767. Assigning a value to **RANDOM** initializes (seeds) the sequence of random numbers. Seeding the random number generator with the same constant value produces the same sequence of values. If **RANDOM** is unset, it loses its special properties, even if it is subsequently reset.

READLINE_ARGUMENT
Any numeric argument given to a **readline** command that was defined using "bind −x" (see **SHELL BUILTIN COMMANDS** below) when it was invoked.

READLINE_LINE
The contents of the **readline** line buffer, for use with "bind −x" (see **SHELL BUILTIN COMMANDS** below).

READLINE_MARK
The position of the mark (saved insertion point) in the **readline** line buffer, for use with "bind −x" (see **SHELL BUILTIN COMMANDS** below). The characters between the insertion point and the

mark are often called the *region*.

READLINE_POINT
The position of the insertion point in the **readline** line buffer, for use with “bind −x” (see **SHELL BUILTIN COMMANDS** below).

REPLY
Set to the line of input read by the **read** builtin command when no arguments are supplied.

SECONDS
Each time this parameter is referenced, it expands to the number of seconds since shell invocation. If a value is assigned to **SECONDS**, the value returned upon subsequent references is the number of seconds since the assignment plus the value assigned. The number of seconds at shell invocation and the current time are always determined by querying the system clock at one-second resolution. If **SECONDS** is unset, it loses its special properties, even if it is subsequently reset.

SHELLOPTS
A colon-separated list of enabled shell options. Each word in the list is a valid argument for the **−o** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below). The options appearing in **SHELLOPTS** are those reported as *on* by **set −o**. If this variable is in the environment when **bash** starts up, the shell enables each option in the list before reading any startup files. If this variable is exported, child shells will enable each option in the list. This variable is read-only.

SHLVL
Incremented by one each time an instance of **bash** is started.

SRANDOM
Each time it is referenced, this variable expands to a 32-bit pseudo-random number. The random number generator is not linear on systems that support */dev/urandom* or *arc4random*(3), so each returned number has no relationship to the numbers preceding it. The random number generator cannot be seeded, so assignments to this variable have no effect. If **SRANDOM** is unset, it loses its special properties, even if it is subsequently reset.

UID Expands to the user ID of the current user, initialized at shell startup. This variable is readonly.

The shell uses the following variables. In some cases, **bash** assigns a default value to a variable; these cases are noted below.

BASH_COMPAT
The value is used to set the shell’s compatibility level. See **SHELL COMPATIBILITY MODE** below for a description of the various compatibility levels and their effects. The value may be a decimal number (e.g., 4.2) or an integer (e.g., 42) corresponding to the desired compatibility level. If **BASH_COMPAT** is unset or set to the empty string, the compatibility level is set to the default for the current version. If **BASH_COMPAT** is set to a value that is not one of the valid compatibility levels, the shell prints an error message and sets the compatibility level to the default for the current version. A subset of the valid values correspond to the compatibility levels described below under **SHELL COMPATIBILITY MODE**. For example, 4.2 and 42 are valid values that correspond to the **compat42 shopt** option and set the compatibility level to 42. The current version is also a valid value.

BASH_ENV
If this parameter is set when **bash** is executing a shell script, its expanded value is interpreted as a filename containing commands to initialize the shell before it reads and executes commands from the script. The value of **BASH_ENV** is subjected to parameter expansion, command substitution, and arithmetic expansion before being interpreted as a filename. **PATH** is not used to search for the resultant filename.

BASH_XTRACEFD
If set to an integer corresponding to a valid file descriptor, **bash** writes the trace output generated when “set −x” is enabled to that file descriptor, instead of the standard error. The file descriptor is closed when **BASH_XTRACEFD** is unset or assigned a new value. Unsetting **BASH_XTRACEFD** or assigning it the empty string causes the trace output to be sent to the standard error. Note that setting **BASH_XTRACEFD** to 2 (the standard error file descriptor) and then unsetting it will result in the standard error being closed.

CDPATH
The search path for the **cd** command. This is a colon-separated list of directories where the shell looks for directories specified as arguments to the **cd** command. A sample value is ".:~:/usr".

CHILD_MAX
Set the number of exited child status values for the shell to remember. **Bash** will not allow this value to be decreased below a POSIX-mandated minimum, and there is a maximum value (currently 8192) that this may not exceed. The minimum value is system-dependent.

COLUMNS
Used by the **select** compound command to determine the terminal width when printing selection lists. Automatically set if the **checkwinsize** option is enabled or in an interactive shell upon receipt of a **SIGWINCH**.

COMPREPLY
An array variable from which **bash** reads the possible completions generated by a shell function invoked by the programmable completion facility (see **Programmable Completion** below). Each array element contains one possible completion.

EMACS
If **bash** finds this variable in the environment when the shell starts with value "t", it assumes that the shell is running in an Emacs shell buffer and disables line editing.

ENV Expanded and executed similarly to **BASH_ENV** (see **INVOCATION** above) when an interactive shell is invoked in posix mode.

EXECIGNORE
A colon-separated list of shell patterns (see **Pattern Matching**) defining the set of filenames to be ignored by command search using **PATH**. Files whose full pathnames match one of these patterns are not considered executable files for the purposes of completion and command execution via **PATH** lookup. This does not affect the behavior of the **[**, **test**, and **[[** commands. Full pathnames in the command hash table are not subject to **EXECIGNORE**. Use this variable to ignore shared library files that have the executable bit set, but are not executable files. The pattern matching honors the setting of the **extglob** shell option.

FCEDIT
The default editor for the **fc** builtin command.

FIGNORE
A colon-separated list of suffixes to ignore when performing filename completion (see **READLINE** below). A filename whose suffix matches one of the entries in **FIGNORE** is excluded from the list of matched filenames. A sample value is ".o:~".

FUNCNEST
If set to a numeric value greater than 0, defines a maximum function nesting level. Function invocations that exceed this nesting level cause the current command to abort.

GLOBIGNORE
A colon-separated list of patterns defining the set of file names to be ignored by pathname expansion. If a file name matched by a pathname expansion pattern also matches one of the patterns in **GLOBIGNORE**, it is removed from the list of matches. The pattern matching honors the setting of the **extglob** shell option.

GLOBSORT
Controls how the results of pathname expansion are sorted. The value of this variable specifies the sort criteria and sort order for the results of pathname expansion. If this variable is unset or set to the null string, pathname expansion uses the historical behavior of sorting by name, in ascending lexicographic order as determined by the **LC_COLLATE** shell variable.

If set, a valid value begins with an optional +, which is ignored, or −, which reverses the sort order from ascending to descending, followed by a sort specifier. The valid sort specifiers are *name*, *numeric*, *size*, *mtime*, *atime*, *ctime*, and *blocks*, which sort the files on name, names in numeric rather than lexicographic order, file size, modification time, access time, inode change time, and number of blocks, respectively. If any of the non-name keys compare as equal (e.g., if two files are the same size), sorting uses the name as a secondary sort key.

For example, a value of *−mtime* sorts the results in descending order by modification time (newest first).

The *numeric* specifier treats names consisting solely of digits as numbers and sorts them using their numeric value (so "2" sorts before "10", for example). When using *numeric*, names containing non-digits sort after all the all-digit names and are sorted by name using the traditional behavior.

A sort specifier of *nosort* disables sorting completely; **bash** returns the results in the order they are read from the file system, ignoring any leading **−**.

If the sort specifier is missing, it defaults to *name*, so a value of + is equivalent to the null string, and a value of - sorts by name in descending order. Any invalid value restores the historical sorting behavior.

HISTCONTROL

A colon-separated list of values controlling how commands are saved on the history list. If the list of values includes *ignorespace*, lines which begin with a **space** character are not saved in the history list. A value of *ignoredups* causes lines matching the previous history entry not to be saved. A value of *ignoreboth* is shorthand for *ignorespace* and *ignoredups*. A value of *erasedups* causes all previous lines matching the current line to be removed from the history list before that line is saved. Any value not in the above list is ignored. If **HISTCONTROL** is unset, or does not include a valid value, **bash** saves all lines read by the shell parser on the history list, subject to the value of **HISTIGNORE**. If the first line of a multi-line compound command was saved, the second and subsequent lines are not tested, and are added to the history regardless of the value of **HISTCONTROL**. If the first line was not saved, the second and subsequent lines of the command are not saved either.

HISTFILE

The name of the file in which command history is saved (see **HISTORY** below). **Bash** assigns a default value of *~/.bash_history*. If **HISTFILE** is unset or null, the shell does not save the command history when it exits.

HISTFILESIZE

The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than the number of history entries that total no more than that number of lines by removing the oldest entries. If the history list contains multi-line entries, the history file may contain more lines than this maximum to avoid leaving partial history entries. The history file is also truncated to this size after writing it when a shell exits or by the **history** builtin. If the value is 0, the history file is truncated to zero size. Non-numeric values and numeric values less than zero inhibit truncation. The shell sets the default value to the value of **HISTSIZE** after reading any startup files.

HISTIGNORE

A colon-separated list of patterns used to decide which command lines should be saved on the history list. If a command line matches one of the patterns in the value of **HISTIGNORE**, it is not saved on the history list. Each pattern is anchored at the beginning of the line and must match the complete line (**bash** does not implicitly append a "*****"). Each pattern is tested against the line after the checks specified by **HISTCONTROL** are applied. In addition to the normal shell pattern matching characters, "**&**" matches the previous history line. A backslash escapes the "**&**"; the backslash is removed before attempting a match. If the first line of a multi-line compound command was saved, the second and subsequent lines are not tested, and are added to the history regardless of the value of **HISTIGNORE**. If the first line was not saved, the second and subsequent lines of the command are not saved either. The pattern matching honors the setting of the **extglob** shell option.

HISTIGNORE subsumes some of the function of **HISTCONTROL**. A pattern of "&" is identical to "ignoredups", and a pattern of "[ ]*" is identical to "ignorespace". Combining these two patterns, separating them with a colon, provides the functionality of "ignoreboth".

HISTSIZE
The number of commands to remember in the command history (see **HISTORY** below). If the value is 0, commands are not saved in the history list. Numeric values less than zero result in every command being saved on the history list (there is no limit). The shell sets the default value to 500 after reading any startup files.

HISTTIMEFORMAT
If this variable is set and not null, its value is used as a format string for *strftime*(3) to print the time stamp associated with each history entry displayed by the **history** builtin. If this variable is set, the shell writes time stamps to the history file so they may be preserved across shell sessions. This uses the history comment character to distinguish timestamps from other history lines.

HOME
The home directory of the current user; the default argument for the **cd** builtin command. The value of this variable is also used when performing tilde expansion.

HOSTFILE
Contains the name of a file in the same format as */etc/hosts* that should be read when the shell needs to complete a hostname. The list of possible hostname completions may be changed while the shell is running; the next time hostname completion is attempted after the value is changed, **bash** adds the contents of the new file to the existing list. If **HOSTFILE** is set, but has no value, or does not name a readable file, **bash** attempts to read */etc/hosts* to obtain the list of possible hostname completions. When **HOSTFILE** is unset, **bash** clears the hostname list.

IFS
The *Internal Field Separator* that is used for word splitting after expansion and to split lines into words with the **read** builtin command. Word splitting is described below under **EXPANSION**. The default value is “<space><tab><newline>”.

IGNOREEOF
Controls the action of an interactive shell on receipt of an **EOF** character as the sole input. If set, the value is the number of consecutive **EOF** characters which must be typed as the first characters on an input line before **bash** exits. If the variable is set but does not have a numeric value, or the value is null, the default value is 10. If it is unset, **EOF** signifies the end of input to the shell.

INPUTRC
The filename for the **readline** startup file, overriding the default of *~/.inputrc* (see **READLINE** below).

INSIDE_EMACS
If this variable appears in the environment when the shell starts, **bash** assumes that it is running inside an Emacs shell buffer and may disable line editing, depending on the value of **TERM**.

LANG
Used to determine the locale category for any category not specifically selected with a variable starting with **LC_**.

LC_ALL
This variable overrides the value of **LANG** and any other **LC_** variable specifying a locale category.

LC_COLLATE
This variable determines the collation order used when sorting the results of pathname expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within pathname expansion and pattern matching.

LC_CTYPE
This variable determines the interpretation of characters and the behavior of character classes within pathname expansion and pattern matching.

LC_MESSAGES
This variable determines the locale used to translate double-quoted strings preceded by a **$**.

LC_NUMERIC
This variable determines the locale category used for number formatting.

LC_TIME
This variable determines the locale category used for data and time formatting.

LINES Used by the **select** compound command to determine the column length for printing selection lists. Automatically set if the **checkwinsize** option is enabled or in an interactive shell upon receipt of a **SIGWINCH**.

MAIL If the value is set to a file or directory name and the **MAILPATH** variable is not set, **bash** informs the user of the arrival of mail in the specified file or Maildir-format directory.

MAILCHECK
Specifies how often (in seconds) **bash** checks for mail. The default is 60 seconds. When it is time to check for mail, the shell does so before displaying the primary prompt. If this variable is unset, or set to a value that is not a number greater than or equal to zero, the shell disables mail checking.

MAILPATH
A colon-separated list of filenames to be checked for mail. The message to be printed when mail arrives in a particular file may be specified by separating the filename from the message with a "?". When used in the text of the message, **$_** expands to the name of the current mailfile. For example:

```
MAILPATH='/var/mail/bfox?"You have mail":~/shell-mail?"$_ has mail!"'
```

Bash can be configured to supply a default value for this variable (there is no value by default), but the location of the user mail files that it uses is system dependent (e.g., /var/mail/**$USER**).

OPTERR
If set to the value 1, **bash** displays error messages generated by the **getopts** builtin command (see **SHELL BUILTIN COMMANDS** below). **OPTERR** is initialized to 1 each time the shell is invoked or a shell script is executed.

PATH The search path for commands. It is a colon-separated list of directories in which the shell looks for commands (see **COMMAND EXECUTION** below). A zero-length (null) directory name in the value of **PATH** indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon. The default path is system-dependent, and is set by the administrator who installs **bash**. A common value is

```
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/bin:/sbin
```

POSIXLY_CORRECT
If this variable is in the environment when **bash** starts, the shell enters posix mode before reading the startup files, as if the **--posix** invocation option had been supplied. If it is set while the shell is running, **bash** enables posix mode, as if the command "set -o posix" had been executed. When the shell enters posix mode, it sets this variable if it was not already set.

PROMPT_COMMAND
If this variable is set, and is an array, the value of each set element is executed as a command prior to issuing each primary prompt. If this is set but not an array variable, its value is used as a command to execute instead.

PROMPT_DIRTRIM
If set to a number greater than zero, the value is used as the number of trailing directory components to retain when expanding the **\w** and **\W** prompt string escapes (see **PROMPTING** below). Characters removed are replaced with an ellipsis.

PS0 The value of this parameter is expanded (see **PROMPTING** below) and displayed by interactive shells after reading a command and before the command is executed.

PS1 The value of this parameter is expanded (see **PROMPTING** below) and used as the primary prompt string. The default value is "\s-\v\$ ".

PS2 The value of this parameter is expanded as with **PS1** and used as the secondary prompt string. The default is "> ".

PS3 The value of this parameter is used as the prompt for the **select** command (see **SHELL GRAMMAR** above).

PS4 The value of this parameter is expanded as with **PS1** and the value is printed before each command **bash** displays during an execution trace. The first character of the expanded value of **PS4** is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is "+ ".

SHELL
This variable expands to the full pathname to the shell. If it is not set when the shell starts, **bash** assigns to it the full pathname of the current user's login shell.

TIMEFORMAT

The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the **time** reserved word should be displayed. The **%** character introduces an escape sequence that is expanded to a time value or other information. The escape sequences and their meanings are as follows; the brackets denote optional portions.

%% A literal **%**.
%[*p*][l]R The elapsed time in seconds.
%[*p*][l]U The number of CPU seconds spent in user mode.
%[*p*][l]S The number of CPU seconds spent in system mode.
%P The CPU percentage, computed as (%U + %S) / %R.

The optional *p* is a digit specifying the *precision*, the number of fractional digits after a decimal point. A value of 0 causes no decimal point or fraction to be output. **time** prints at most six digits after the decimal point; values of *p* greater than 6 are changed to 6. If *p* is not specified, **time** prints three digits after the decimal point.

The optional **l** specifies a longer format, including minutes, of the form *MM*m*SS*.*FF*s. The value of *p* determines whether or not the fraction is included.

If this variable is not set, **bash** acts as if it had the value **$'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'**. If the value is null, **bash** does not display any timing information. A trailing newline is added when the format string is displayed.

TMOUT

If set to a value greater than zero, the **read** builtin uses the value as its default timeout. The **select** command terminates if input does not arrive after **TMOUT** seconds when input is coming from a terminal. In an interactive shell, the value is interpreted as the number of seconds to wait for a line of input after issuing the primary prompt. **Bash** terminates after waiting for that number of seconds if a complete line of input does not arrive.

TMPDIR

If set, **bash** uses its value as the name of a directory in which **bash** creates temporary files for the shell's use.

auto_resume

This variable controls how the shell interacts with the user and job control. If this variable is set, simple commands consisting of only a single word, without redirections, are treated as candidates for resumption of an existing stopped job. There is no ambiguity allowed; if there is more than one job beginning with or containing the word, this selects the most recently accessed job. The *name* of a stopped job, in this context, is the command line used to start it, as displayed by **jobs**. If set to the value *exact*, the word must match the name of a stopped job exactly; if set to *substring*, the word needs to match a substring of the name of a stopped job. The *substring* value provides functionality analogous to the **%?** job identifier (see **JOB CONTROL** below). If set to any other value (e.g., *prefix*), the word must be a prefix of a stopped job's name; this provides functionality analogous to the **%***string* job identifier.

histchars

The two or three characters which control history expansion, quick substitution, and tokenization (see **HISTORY EXPANSION** below). The first character is the *history expansion* character, the character which begins a history expansion, normally "**!**". The second character is the *quick substitution* character, normally "**^**". When it appears as the first character on the line, history substitution repeats the previous command, replacing one string with another. The optional third character is the *history comment* character, normally "**#**", which indicates that the remainder of the line is a comment when it appears as the first character of a word. The history comment character disables history substitution for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

Arrays

Bash provides one-dimensional indexed and associative array variables. Any variable may be used as an indexed array; the **declare** builtin explicitly declares an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Indexed arrays are referenced using arithmetic expressions that must expand to an integer (see **ARITHMETIC EVALUATION** below) and are zero-based; associative arrays are referenced using arbitrary strings. Unless otherwise noted, indexed array indices must be non-negative integers.

The shell performs parameter and variable expansion, arithmetic expansion, command substitution, and quote removal on indexed array subscripts. Since this can potentially result in empty strings, subscript indexing treats those as expressions that evaluate to 0.

The shell performs tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, and quote removal on associative array subscripts. Empty strings cannot be used as associative array keys.

Bash automatically creates an indexed array if any variable is assigned to using the syntax

name[*subscript*]=*value* .

The *subscript* is treated as an arithmetic expression that must evaluate to a number greater than or equal to zero. To explicitly declare an indexed array, use

declare −a *name*

(see **SHELL BUILTIN COMMANDS** below).

declare −a *name*[*subscript*]

is also accepted; the *subscript* is ignored.

Associative arrays are created using

declare −A *name*

.

Attributes may be specified for an array variable using the **declare** and **readonly** builtins. Each attribute applies to all members of an array.

Arrays are assigned using compound assignments of the form *name*=(value*1* ... value*n*), where each *value* may be of the form [*subscript*]=*string*. Indexed array assignments do not require anything but *string*. Each *value* in the list is expanded using the shell expansions described below under **EXPANSION**, but *value*s that are valid variable assignments including the brackets and subscript do not undergo brace expansion and word splitting, as with individual variable assignments.

When assigning to indexed arrays, if the optional brackets and subscript are supplied, that index is assigned to; otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero.

When assigning to an associative array, the words in a compound assignment may be either assignment statements, for which the subscript is required, or a list of words that is interpreted as a sequence of alternating keys and values: *name*=(*key1 value1 key2 value2* ...). These are treated identically to *name*=([*key1*]=*value1* [*key2*]=*value2* ...). The first word in the list determines how the remaining words are interpreted; all assignments in a list must be of the same type. When using key/value pairs, the keys may not be missing or empty; a final missing value is treated like the empty string.

This syntax is also accepted by the **declare** builtin. Individual array elements may be assigned to using the *name*[*subscript*]=*value* syntax introduced above.

When assigning to an indexed array, if *name* is subscripted by a negative number, that number is interpreted as relative to one greater than the maximum index of *name*, so negative indices count back from the end of the array, and an index of −1 references the last element.

The “+=” operator appends to an array variable when assigning using the compound assignment syntax; see **PARAMETERS** above.

An array element is referenced using ${*name*[*subscript*]}. The braces are required to avoid conflicts with pathname expansion. If *subscript* is @ or *, the word expands to all members of *name*, unless noted in the description of a builtin or word expansion. These subscripts differ only when the word appears within double quotes. If the word is double-quoted, ${*name*[*]} expands to a single word with the value of each array member separated by the first character of the **IFS** special variable, and ${*name*[@]} expands each element of *name* to a separate word. When there are no array members, ${*name*[@]} expands to nothing. If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the expansion of the original word, and the expansion of the last parameter is joined with the last part of the expansion of the original word. This is analogous to the expansion of the special parameters ***** and **@** (see **Special Parameters** above).

${#*name*[*subscript*]} expands to the length of ${*name*[*subscript*]}. If *subscript* is ***** or **@**, the expansion is the number of elements in the array.

If the *subscript* used to reference an element of an indexed array evaluates to a number less than zero, it is interpreted as relative to one greater than the maximum index of the array, so negative indices count back from the end of the array, and an index of −1 references the last element.

Referencing an array variable without a subscript is equivalent to referencing the array with a subscript of 0. Any reference to a variable using a valid subscript is valid; **bash** creates an array if necessary.

An array variable is considered set if a subscript has been assigned a value. The null string is a valid value.

It is possible to obtain the keys (indices) of an array as well as the values. ${**!***name*[@]} and ${**!***name*[*]} expand to the indices assigned in array variable *name*. The treatment when in double quotes is similar to the expansion of the special parameters @ and * within double quotes.

The **unset** builtin is used to destroy arrays. **unset** *name*[*subscript*] unsets the array element at index *subscript*, for both indexed and associative arrays. Negative subscripts to indexed arrays are interpreted as described above. Unsetting the last element of an array variable does not unset the variable. **unset** *name*, where *name* is an array, removes the entire array. **unset** *name*[*subscript*] behaves differently depending on whether *name* is an indexed or associative array when *subscript* is ***** or **@**. If *name* is an associative array, this unsets the element with subscript ***** or **@**. If *name* is an indexed array, unset removes all of the elements but does not remove the array itself.

When using a variable name with a subscript as an argument to a command, such as with **unset**, without using the word expansion syntax described above, (e.g., unset a[4]), the argument is subject to pathname expansion. Quote the argument if pathname expansion is not desired (e.g., unset 'a[4]').

The **declare**, **local**, and **readonly** builtins each accept a **−a** option to specify an indexed array and a **−A** option to specify an associative array. If both options are supplied, **−A** takes precedence. The **read** builtin accepts a **−a** option to assign a list of words read from the standard input to an array. The **set** and **declare** builtins display array values in a way that allows them to be reused as assignments. Other builtins accept array name arguments as well (e.g., **mapfile**); see the descriptions of individual builtins below for details. The shell provides a number of builtin array variables.

EXPANSION

Expansion is performed on the command line after it has been split into words. The shell performs these expansions: *brace expansion*, *tilde expansion*, *parameter and variable expansion*, *command substitution*, *arithmetic expansion*, *word splitting*, *pathname expansion*, and *quote removal*.

The order of expansions is: brace expansion; tilde expansion, parameter and variable expansion, arithmetic expansion, and command substitution (done in a left-to-right fashion); word splitting; pathname expansion; and quote removal.

On systems that can support it, there is an additional expansion available: *process substitution*. This is performed at the same time as tilde, parameter, variable, and arithmetic expansion and command substitution.

Quote removal is always performed last. It removes quote characters present in the original word, not ones resulting from one of the other expansions, unless they have been quoted themselves.

Only brace expansion, word splitting, and pathname expansion can increase the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of **"$@"** and **"${***name***[@]}"**, and, in most cases, **$*** and **${***name***[*]}** as explained above (see **PARAMETERS**).

Brace Expansion

Brace expansion is a mechanism to generate arbitrary strings sharing a common prefix and suffix, either of which can be empty. This mechanism is similar to *pathname expansion*, but the filenames generated need not exist. Patterns to be brace expanded are formed from an optional *preamble*, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional *postscript*. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; brace expansion preserves left to right order. For example, a**{**d,c,b**}**e expands into “ade ace abe”.

A sequence expression takes the form ***x*..*y*[..*incr*]**, where *x* and *y* are either integers or single letters, and *incr*, an optional increment, is an integer. When integers are supplied, the expression expands to each number between *x* and *y*, inclusive. If either *x* or *y* begins with a zero, each generated term will contain the same number of digits, zero-padding where necessary. When letters are supplied, the expression expands to each character lexicographically between *x* and *y*, inclusive, using the C locale. Note that both *x* and *y* must be of the same type (integer or letter). When the increment is supplied, it is used as the difference between each term. The default increment is 1 or −1 as appropriate.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. **Bash** does not apply any syntactic interpretation to the context of the expansion or the text between the braces.

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma or a valid sequence expression. Any incorrectly formed brace expansion is left unchanged.

A “{” or Q , may be quoted with a backslash to prevent its being considered part of a brace expression. To avoid conflicts with parameter expansion, the string “${” is not considered eligible for brace expansion, and inhibits brace expansion until the closing “}”.

This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example:

```
mkdir /usr/local/src/bash/{old,new,dist,bugs}
```

or

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

Brace expansion introduces a slight incompatibility with historical versions of **sh**. **sh** does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. **Bash** removes braces from words as a consequence of brace expansion. For example, a word entered to **sh** as “file{1,2}” appears identically in the output. **Bash** outputs that word as “file1 file2” after brace expansion. Start **bash** with the **+B** option or disable brace expansion with the **+B** option to the **set** command (see **SHELL BUILTIN COMMANDS** below) for strict **sh** compatibility.

Tilde Expansion

If a word begins with an unquoted tilde character (“**~**”), all of the characters preceding the first unquoted slash (or all characters, if there is no unquoted slash) are considered a *tilde-prefix*. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible *login name*. If this login name is the null string, the tilde is replaced with the value of the shell parameter **HOME**. If **HOME** is unset, the tilde expands to the home directory of the user executing the shell instead. Otherwise, the tilde-prefix is replaced with the home directory associated with the specified login name.

If the tilde-prefix is a “~+”, the value of the shell variable **PWD** replaces the tilde-prefix. If the tilde-prefix is a “~−”, the shell substitutes the value of the shell variable **OLDPWD**, if it is set. If the characters following the tilde in the tilde-prefix consist of a number *N*, optionally prefixed by a “+” or a “−”, the tilde-prefix is replaced with the corresponding element from the directory stack, as it would be displayed by the **dirs** builtin invoked with the characters following the tilde in the tilde-prefix as an argument. If the characters following the tilde in the tilde-prefix consist of a number without a leading “+” or “−”, tilde expansion assumes “+”.

The results of tilde expansion are treated as if they were quoted, so the replacement is not subject to word splitting and pathname expansion.

If the login name is invalid, or the tilde expansion fails, the tilde-prefix is unchanged.

Bash checks each variable assignment for unquoted tilde-prefixes immediately following a **:** or the first **=**, and performs tilde expansion in these cases. Consequently, one may use filenames with tildes in assignments to **PATH**, **MAILPATH**, and **CDPATH**, and the shell assigns the expanded value.

Bash also performs tilde expansion on words satisfying the conditions of variable assignments (as described above under **PARAMETERS**) when they appear as arguments to simple commands. **Bash** does not do this, except for the *declaration* commands listed above, when in posix mode.

Parameter Expansion

The "**$**" character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

When braces are used, the matching ending brace is the first "**}**" not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion.

The basic form of parameter expansion is

${*parameter*}

which substitutes the value of *parameter*. The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character which is not to be interpreted as part of its name. The *parameter* is a shell parameter as described above **PARAMETERS**) or an array reference (**Arrays**).

If the first character of *parameter* is an exclamation point (**!**), and *parameter* is not a *nameref*, it introduces a level of indirection. **Bash** uses the value formed by expanding the rest of *parameter* as the new *parameter*; this new parameter is then expanded and that value is used in the rest of the expansion, rather than the expansion of the original *parameter*. This is known as *indirect expansion*. The value is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *parameter* is a nameref, this expands to the name of the parameter referenced by *parameter* instead of performing the complete indirect expansion, for compatibility. The exceptions to this are the expansions of ${**!***prefix******} and ${**!***name*[*@*]} described below. The exclamation point must immediately follow the left brace in order to introduce indirection.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion.

When not performing substring expansion, using the forms documented below (e.g., **:-**), **bash** tests for a parameter that is unset or null. Omitting the colon tests only for a parameter that is unset.

${*parameter***:−***word*}
: **Use Default Values**. If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.

${*parameter***:=***word*}
: **Assign Default Values**. If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*, and the expansion is the final value of *parameter*. Positional parameters and special parameters may not be assigned in this way.

${*parameter***:?***word*}
: **Display Error if Null or Unset**. If *parameter* is null or unset, the shell writes the expansion of *word* (or a message to that effect if *word* is not present) to the standard error and, if it is not interactive, exits with a non-zero status. An interactive shell does not exit, but does not execute the command associated with the expansion. Otherwise, the value of *parameter* is substituted.

${*parameter***:+***word*}
: **Use Alternate Value**. If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted. The value of *parameter* is not used.

${*parameter***:***offset*}
${*parameter***:***offset***:***length*}

Substring Expansion. Expands to up to *length* characters of the value of *parameter* starting at the character specified by *offset*. If *parameter* is @ or *, an indexed array subscripted by @ or *, or an associative array name, the results differ as described below. If **:***length* is omitted (the first form above), this expands to the substring of the value of *parameter* starting at the character specified by *offset* and extending to the end of the value. If *offset* is omitted, it is treated as 0. If *length* is omitted, but the colon after *offset* is present, it is treated as 0. *length* and *offset* are arithmetic expressions (see **ARITHMETIC EVALUATION** below).

If *offset* evaluates to a number less than zero, the value is used as an offset in characters from the end of the value of *parameter*. If *length* evaluates to a number less than zero, it is interpreted as an offset in characters from the end of the value of *parameter* rather than a number of characters, and the expansion is the characters between *offset* and that result. Note that a negative offset must be separated from the colon by at least one space to avoid being confused with the **:-** expansion.

If *parameter* is @ or *, the result is *length* positional parameters beginning at *offset*. A negative *offset* is taken relative to one greater than the greatest positional parameter, so an offset of −1 evaluates to the last positional parameter (or 0 if there are no positional parameters). It is an expansion error if *length* evaluates to a number less than zero.

If *parameter* is an indexed array name subscripted by @ or *, the result is the *length* members of the array beginning with ${*parameter*[*offset*]}. A negative *offset* is taken relative to one greater than the maximum index of the specified array. It is an expansion error if *length* evaluates to a number less than zero.

Substring expansion applied to an associative array produces undefined results.

Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1 by default. If *offset* is 0, and the positional parameters are used, **$0** is prefixed to the list.

${**!***prefix******}
${**!***prefix*@}

Names matching prefix. Expands to the names of variables whose names begin with *prefix*, separated by the first character of the **IFS** special variable. When @ is used and the expansion appears within double quotes, each variable name expands to a separate word.

${**!***name*[@]}
${**!***name*[*]}

List of array keys. If *name* is an array variable, expands to the list of array indices (keys) assigned in *name*. If *name* is not an array, expands to 0 if *name* is set and null otherwise. When @ is used and the expansion appears within double quotes, each key expands to a separate word.

${**#***parameter*}

Parameter length. Substitutes the length in characters of the expanded value of *parameter*. If *parameter* is * or @, the value substituted is the number of positional parameters. If *parameter* is an array name subscripted by * or @, the value substituted is the number of elements in the array. If *parameter* is an indexed array name subscripted by a negative number, that number is interpreted as relative to one greater than the maximum index of *parameter*, so negative indices count back from the end of the array, and an index of −1 references the last element.

${*parameter***#***word*}
${*parameter***##***word*}

Remove matching prefix pattern. The *word* is expanded to produce a pattern just as in pathname expansion, and matched against the expanded value of *parameter* using the rules described under **Pattern Matching** below. If the pattern matches the beginning of the value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the "#" case) or the longest matching pattern (the "##" case) deleted. If *parameter* is @ or *, the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

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

Remove matching suffix pattern. The *word* is expanded to produce a pattern just as in pathname expansion, and matched against the expanded value of *parameter* using the rules described under **Pattern Matching** below. If the pattern matches a trailing portion of the expanded value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the "%" case) or the longest matching pattern (the "%%" case) deleted. If *parameter* is @ or *, the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter***/***pattern***/***string*}
${*parameter***//***pattern***/***string*}
${*parameter***/#***pattern***/***string*}
${*parameter***/%***pattern***/***string*}

Pattern substitution. The *pattern* is expanded to produce a pattern and matched against the expanded value of *parameter* as described under **Pattern Matching** below. The longest match of *pattern* in the expanded value is replaced with *string*. *string* undergoes tilde expansion, parameter and variable expansion, arithmetic expansion, command and process substitution, and quote removal.

In the first form above, only the first match is replaced. If there are two slashes separating *parameter* and *pattern* (the second form above), all matches of *pattern* are replaced with *string*. If *pattern* is preceded by **#** (the third form above), it must match at the beginning of the expanded value of *parameter*. If *pattern* is preceded by **%** (the fourth form above), it must match at the end of the expanded value of *parameter*.

If the expansion of *string* is null, matches of *pattern* are deleted and the **/** following *pattern* may be omitted.

If the **patsub_replacement** shell option is enabled using **shopt**, any unquoted instances of **&** in *string* are replaced with the matching portion of *pattern*.

Quoting any part of *string* inhibits replacement in the expansion of the quoted portion, including replacement strings stored in shell variables. Backslash escapes **&** in *string*; the backslash is removed in order to permit a literal **&** in the replacement string. Backslash can also be used to escape a backslash; **** results in a literal backslash in the replacement. Users should take care if *string* is double-quoted to avoid unwanted interactions between the backslash and double-quoting, since backslash has special meaning within double quotes. Pattern substitution performs the check for unquoted **&** after expanding *string*; shell programmers should quote any occurrences of **&** they want to be taken literally in the replacement and ensure any instances of **&** they want to be replaced are unquoted.

Like the pattern removal operators, double quotes surrounding the replacement string quote the expanded characters, while double quotes enclosing the entire parameter substitution do not, since the expansion is performed in a context that doesn't take any enclosing double quotes into account.

If the **nocasematch** shell option is enabled, the match is performed without regard to the case of alphabetic characters.

If *parameter* is @ or *, the substitution operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the substitution operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter*^*pattern*}
${*parameter*^^*pattern*}
${*parameter*,*pattern*}
${*parameter*,,*pattern*}

Case modification. This expansion modifies the case of alphabetic characters in *parameter*. First, the *pattern* is expanded to produce a pattern as described below under **Pattern Matching**. **Bash** then examines characters in the expanded value of *parameter* against *pattern* as described below. If a character matches the pattern, its case is converted. The pattern should not attempt to match more than one character.

Using "^" converts lowercase letters matching *pattern* to uppercase; "," converts matching uppercase letters to lowercase. The **^** and **,** variants examine the first character in the expanded value and convert its case if it matches *pattern*; the **^^** and **,,** variants examine all characters in the expanded value and convert each one that matches *pattern*. If *pattern* is omitted, it is treated like a **?**, which matches every character.

If *parameter* is @ or *, the case modification operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the case modification operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter*@*operator*}

Parameter transformation. The expansion is either a transformation of the value of *parameter* or information about *parameter* itself, depending on the value of *operator*. Each *operator* is a single letter:

- **U** The expansion is a string that is the value of *parameter* with lowercase alphabetic characters converted to uppercase.
- **u** The expansion is a string that is the value of *parameter* with the first character converted to uppercase, if it is alphabetic.
- **L** The expansion is a string that is the value of *parameter* with uppercase alphabetic characters converted to lowercase.
- **Q** The expansion is a string that is the value of *parameter* quoted in a format that can be reused as input.
- **E** The expansion is a string that is the value of *parameter* with backslash escape sequences expanded as with the **$'**...**'** quoting mechanism.
- **P** The expansion is a string that is the result of expanding the value of *parameter* as if it were a prompt string (see **PROMPTING** below).
- **A** The expansion is a string in the form of an assignment statement or **declare** command that, if evaluated, recreates *parameter* with its attributes and value.
- **K** Produces a possibly-quoted version of the value of *parameter*, except that it prints the values of indexed and associative arrays as a sequence of quoted key-value pairs (see **Arrays** above). The keys and values are quoted in a format that can be reused as input.
- **a** The expansion is a string consisting of flag values representing *parameter*'s attributes.
- **k** Like the K transformation, but expands the keys and values of indexed and associative arrays to separate words after word splitting.

If *parameter* is @ or *, the operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the operation is applied to each member of the array in turn, and the expansion is the resultant list.

The result of the expansion is subject to word splitting and pathname expansion as described below.

Command Substitution

Command substitution allows the output of a command to replace the command itself. There are two standard forms:

$(*command* **)**

or (deprecated)

`***command***`.

Bash performs the expansion by executing *command* in a subshell environment and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting. The command substitution **$(cat** *file***)** can be replaced by the equivalent but faster **$(<** *file***)**.

With the old-style backquote form of substitution, backslash retains its literal meaning except when followed by **$**, **`**, or ****. The first backquote not preceded by a backslash terminates the command substitution. When using the $(*command*) form, all characters between the parentheses make up the command; none are treated specially.

There is an alternate form of command substitution:

${*c command***; }**

which executes *command* in the current execution environment and captures its output, again with trailing newlines removed.

The character *c* following the open brace must be a space, tab, newline, or **|**, and the close brace must be in a position where a reserved word may appear (i.e., preceded by a command terminator such as semicolon). **Bash** allows the close brace to be joined to the remaining characters in the word without being followed by a shell metacharacter as a reserved word would usually require.

Any side effects of *command* take effect immediately in the current execution environment and persist in the current environment after the command completes (e.g., the **exit** builtin exits the shell).

This type of command substitution superficially resembles executing an unnamed shell function: local variables are created as when a shell function is executing, and the **return** builtin forces *command* to complete; however, the rest of the execution environment, including the positional parameters, is shared with the caller.

If the first character following the open brace is a **|**, the construct expands to the value of the **REPLY** shell variable after *command* executes, without removing any trailing newlines, and the standard output of *command* remains the same as in the calling shell. **Bash** creates **REPLY** as an initially-unset local variable when *command* executes, and restores **REPLY** to the value it had before the command substitution after *command* completes, as with any local variable.

Command substitutions may be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes.

If the substitution appears within double quotes, **bash** does not perform word splitting and pathname expansion on the results.

Arithmetic Expansion

Arithmetic expansion evaluates an arithmetic expression and substitutes the result. The format for arithmetic expansion is:

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

The *expression* undergoes the same expansions as if it were within double quotes, but unescaped double quote characters in *expression* are not treated specially and are removed. All tokens in the expression undergo parameter and variable expansion, command substitution, and quote removal. The result is treated as the arithmetic expression to be evaluated. Since the way Bash handles double quotes can potentially result in empty strings, arithmetic expansion treats those as expressions that evaluate to 0. Arithmetic expansions may be nested.

The evaluation is performed according to the rules listed below under **ARITHMETIC EVALUATION**. If *expression* is invalid, **bash** prints a message to standard error indicating failure, does not perform the substitution, and does not execute the command associated with the expansion.

Process Substitution

Process substitution allows a process's input or output to be referred to using a filename. It takes the form of **<(***list***)** or **>(***list***)**. The process *list* is run asynchronously, and its input or output appears as a filename.

This filename is passed as an argument to the current command as the result of the expansion.

If the **>(***list***)** form is used, writing to the file provides input for *list*. If the **<(***list***)** form is used, reading the file obtains the output of *list*. No space may appear between the **<** or **>** and the left parenthesis, otherwise the construct would be interpreted as a redirection.

Process substitution is supported on systems that support named pipes (*FIFOs*) or the */dev/fd* method of naming open files.

When available, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

Word Splitting

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for *word splitting*. Words that were not expanded are not split.

The shell treats each character of **IFS** as a delimiter, and splits the results of the other expansions into words using these characters as field terminators.

An *IFS whitespace* character is whitespace as defined above (see **Definitions**) that appears in the value of **IFS**. Space, tab, and newline are always considered IFS whitespace, even if they don't appear in the locale's **space** category.

If **IFS** is unset, field splitting acts as if its value were **<space><tab><newline>**, and treats these characters as IFS whitespace. If the value of **IFS** is null, no word splitting occurs, but implicit null arguments (see below) are still removed.

Word splitting begins by removing sequences of IFS whitespace characters from the beginning and end of the results of the previous expansions, then splits the remaining words.

If the value of **IFS** consists solely of IFS whitespace, any sequence of IFS whitespace characters delimits a field, so a field consists of characters that are not unquoted IFS whitespace, and null fields result only from quoting.

If **IFS** contains a non-whitespace character, then any character in the value of **IFS** that is not IFS whitespace, along with any adjacent IFS whitespace characters, delimits a field. This means that adjacent non-IFS-whitespace delimiters produce a null field. A sequence of IFS whitespace characters also delimits a field.

Explicit null arguments (**""** or **''**) are retained and passed to commands as empty strings. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. Expanding a parameter with no value within double quotes produces a null field, which is retained and passed to a command as an empty string.

When a quoted null argument appears as part of a word whose expansion is non-null, word splitting removes the null argument portion, leaving the non-null expansion. That is, the word “–d''” becomes “–d” after word splitting and null argument removal.

Pathname Expansion

After word splitting, unless the **–f** option has been set, **bash** scans each word for the characters *****, **?**, and **[**. If one of these characters appears, and is not quoted, then the word is regarded as a *pattern*, and replaced with a sorted list of filenames matching the pattern (see **Pattern Matching** below) subject to the value of the **GLOBSORT** shell variable.

If no matching filenames are found, and the shell option **nullglob** is not enabled, the word is left unchanged. If the **nullglob** option is set, and no matches are found, the word is removed. If the **failglob** shell option is set, and no matches are found, **bash** prints an error message and does not execute the command. If the shell option **nocaseglob** is enabled, the match is performed without regard to the case of alphabetic characters.

When a pattern is used for pathname expansion, the character “.” at the start of a name or immediately following a slash must be matched explicitly, unless the shell option **dotglob** is set. In order to match the filenames . and .., the pattern must begin with “.” (for example, “.?”), even if **dotglob** is set. If the

globskipdots shell option is enabled, the filenames . and .. never match, even if the pattern begins with a ".". When not matching pathnames, the "." character is not treated specially.

When matching a pathname, the slash character must always be matched explicitly by a slash in the pattern, but in other matching contexts it can be matched by a special pattern character as described below under **Pattern Matching**.

See the description of **shopt** below under **SHELL BUILTIN COMMANDS** for a description of the **nocaseglob**, **nullglob**, **globskipdots**, **failglob**, and **dotglob** shell options.

The **GLOBIGNORE** shell variable may be used to restrict the set of file names matching a *pattern*. If **GLOBIGNORE** is set, each matching file name that also matches one of the patterns in **GLOBIGNORE** is removed from the list of matches. If the **nocaseglob** option is set, the matching against the patterns in **GLOBIGNORE** is performed without regard to case. The filenames . and .. are always ignored when **GLOBIGNORE** is set and not null. However, setting **GLOBIGNORE** to a non-null value has the effect of enabling the **dotglob** shell option, so all other filenames beginning with a "." match. To get the old behavior of ignoring filenames beginning with a ".", make ".*" one of the patterns in **GLOBIGNORE**. The **dotglob** option is disabled when **GLOBIGNORE** is unset. The **GLOBIGNORE** pattern matching honors the setting of the **extglob** shell option.

The value of the **GLOBSORT** shell variable controls how the results of pathname expansion are sorted, as described above under **Shell Variables**.

Pattern Matching

Any character that appears in a pattern, other than the special pattern characters described below, matches itself. The NUL character may not occur in a pattern. A backslash escapes the following character; the escaping backslash is discarded when matching. The special pattern characters must be quoted if they are to be matched literally.

The special pattern characters have the following meanings:

*****
: Matches any string, including the null string. When the **globstar** shell option is enabled, and ***** is used in a pathname expansion context, two adjacent *****s used as a single pattern match all files and zero or more directories and subdirectories. If followed by a **/**, two adjacent *****s match only directories and subdirectories.

?
: Matches any single character.

[...]
: Matches any one of the characters enclosed between the brackets. This is known as a *bracket expression* and matches a single character. A pair of characters separated by a hyphen denotes a *range expression*; any character that falls between those two characters, inclusive, using the current locale's collating sequence and character set, matches. If the first character following the **[** is a **!** or a **^** then any character not within the range matches. To match a **–**, include it as the first or last character in the set. To match a **]**, include it as the first character in the set.

  The sorting order of characters in range expressions, and the characters included in the range, are determined by the current locale and the values of the **LC_COLLATE** or **LC_ALL** shell variables, if set. To obtain the traditional interpretation of range expressions, where **[a–d]** is equivalent to **[abcd]**, set the value of the **LC_COLLATE** or **LC_ALL** shell variables to **C**, or enable the **globasciiranges** shell option.

  Within a bracket expression, *character classes* can be specified using the syntax **[:***class***:]**, where *class* is one of the following classes defined in the POSIX standard:

  alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit

  A character class matches any character belonging to that class. The **word** character class matches letters, digits, and the character _.

  Within a bracket expression, an *equivalence class* can be specified using the syntax **[=***c***=]**, which matches all characters with the same collation weight (as defined by the current locale) as the character *c*.

Within a bracket expression, the syntax **[.***symbol***.]** matches the collating symbol *symbol*.

If the **extglob** shell option is enabled using the **shopt** builtin, the shell recognizes several extended pattern matching operators. In the following description, a *pattern-list* is a list of one or more patterns separated by a **|**. Composite patterns may be formed using one or more of the following sub-patterns:

?(*pattern-list***)**
Matches zero or one occurrence of the given patterns.

(pattern-list***)**
Matches zero or more occurrences of the given patterns.

+(*pattern-list***)**
Matches one or more occurrences of the given patterns.

@(*pattern-list***)**
Matches one of the given patterns.

!(*pattern-list***)**
Matches anything except one of the given patterns.

The **extglob** option changes the behavior of the parser, since the parentheses are normally treated as operators with syntactic meaning. To ensure that extended matching patterns are parsed correctly, make sure that **extglob** is enabled before parsing constructs containing the patterns, including shell functions and command substitutions.

When matching filenames, the **dotglob** shell option determines the set of filenames that are tested: when **dotglob** is enabled, the set of filenames includes all files beginning with “.”, but . and .. must be matched by a pattern or sub-pattern that begins with a dot; when it is disabled, the set does not include any filenames beginning with “.” unless the pattern or sub-pattern begins with a “.”. If the **globskipdots** shell option is enabled, the filenames . and .. never appear in the set. As above, “.” only has a special meaning when matching filenames.

Complicated extended pattern matching against long strings is slow, especially when the patterns contain alternations and the strings contain multiple matches. Using separate matches against shorter strings, or using arrays of strings instead of a single long string, may be faster.

Quote Removal

After the preceding expansions, all unquoted occurrences of the characters ****, **'**, and **"** that did not result from one of the above expansions are removed.

REDIRECTION

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. *Redirection* allows commands’ file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. When used with the **exec** builtin, redirections modify file handles in the current shell execution environment. The following redirection operators may precede or appear anywhere within a *simple command* or may follow a *command*. Redirections are processed in the order they appear, from left to right.

Each redirection that may be preceded by a file descriptor number may instead be preceded by a word of the form {*varname*}. In this case, for each redirection operator except **>&−** and **<&−**, the shell allocates a file descriptor greater than or equal to 10 and assigns it to *varname*. If {*varname*} precedes **>&−** or **<&−**, the value of *varname* defines the file descriptor to close. If {*varname*} is supplied, the redirection persists beyond the scope of the command, which allows the shell programmer to manage the file descriptor’s lifetime manually without using the **exec** builtin. The **varredir_close** shell option manages this behavior.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is “<”, the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is “>”, the redirection refers to the standard output (file descriptor 1).

The *word* following the redirection operator in the following descriptions, unless otherwise noted, is subjected to brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, quote removal, pathname expansion, and word splitting. If it expands to more than one word, **bash** reports an error.

The order of redirections is significant. For example, the command

ls **>** dirlist 2**>&**1

directs both standard output and standard error to the file *dirlist*, while the command

ls 2**>&**1 **>** dirlist

directs only the standard output to file *dirlist*, because the standard error was directed to the standard output before the standard output was redirected to *dirlist*.

Bash handles several filenames specially when they are used in redirections, as described in the following table. If the operating system on which **bash** is running provides these special files, **bash** uses them; otherwise it emulates them internally with the behavior described below.

/dev/fd/*fd*
: If *fd* is a valid integer, duplicate file descriptor *fd*.

/dev/stdin
: File descriptor 0 is duplicated.

/dev/stdout
: File descriptor 1 is duplicated.

/dev/stderr
: File descriptor 2 is duplicated.

/dev/tcp/*host***/***port*
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, **bash** attempts to open the corresponding TCP socket.

/dev/udp/*host***/***port*
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, **bash** attempts to open the corresponding UDP socket.

A failure to open or create a file causes the redirection to fail.

Redirections using file descriptors greater than 9 should be used with care, as they may conflict with file descriptors the shell uses internally.

Redirecting Input

Redirecting input opens the file whose name results from the expansion of *word* for reading on file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified.

The general format for redirecting input is:

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

Redirecting Output

Redirecting output opens the file whose name results from the expansion of *word* for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

The general format for redirecting output is:

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

If the redirection operator is **>**, and the **noclobber** option to the **set** builtin command has been enabled, the redirection fails if the file whose name results from the expansion of *word* exists and is a regular file. If the redirection operator is **>|**, or the redirection operator is **>** and the **noclobber** option to the **set** builtin is not enabled, **bash** attempts the redirection even if the file named by *word* exists.

Appending Redirected Output

Redirecting output in this fashion opens the file whose name results from the expansion of *word* for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

The general format for appending output is:

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

Redirecting Standard Output and Standard Error

This construct redirects both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to the file whose name is the expansion of *word*.

There are two formats for redirecting standard output and standard error:

&>*word*

and

>&*word*

Of the two forms, the first is preferred. This is semantically equivalent to

>*word* 2**>&**1

When using the second form, *word* may not expand to a number or **−**. If it does, other redirection operators apply (see **Duplicating File Descriptors** below) for compatibility reasons.

Appending Standard Output and Standard Error

This construct appends both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to the file whose name is the expansion of *word*.

The format for appending standard output and standard error is:

&>>*word*

This is semantically equivalent to

>>*word* 2**>&**1

(see **Duplicating File Descriptors** below).

Here Documents

This type of redirection instructs the shell to read input from the current source until it reads a line containing only *delimiter* (with no trailing blanks). All of the lines read up to that point then become the standard input (or file descriptor *n* if *n* is specified) for a command.

The format of here-documents is:

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

The shell does not perform parameter and variable expansion, command substitution, arithmetic expansion, or pathname expansion on *word*.

If any part of *word* is quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. If *word* is unquoted, the *delimiter* is *word* itself, and the here-document text is treated similarly to a double-quoted string: all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion, the character sequence **\<newline>** is treated literally, and **** must be used to quote the characters ****, **$**, and **`**; however, double quote characters have no special meaning.

If the redirection operator is **<<−**, then the shell strips all leading tab characters from input lines and the line containing *delimiter*. This allows here-documents within shell scripts to be indented in a natural fashion.

If the delimiter is not quoted, the **\<newline>** sequence is treated as a line continuation: the two lines are joined and the backslash-newline is removed. This happens while reading the here-document, before the check for the ending delimiter, so joined lines can form the end delimiter.

Here Strings

A variant of here documents, the format is:

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

The *word* undergoes tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion and word splitting are not performed. The result is supplied as a single string, with a newline appended, to the command on its standard input (or file descriptor *n* if *n* is specified).

Duplicating File Descriptors

The redirection operator

[*n*]**<&***word*

is used to duplicate input file descriptors. If *word* expands to one or more digits, file descriptor *n* is made to be a copy of that file descriptor. It is a redirection error if the digits in *word* do not specify a file descriptor open for input. If *word* evaluates to **–**, file descriptor *n* is closed. If *n* is not specified, this uses the standard input (file descriptor 0).

The operator

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

is used similarly to duplicate output file descriptors. If *n* is not specified, this uses the standard output (file descriptor 1). It is a redirection error if the digits in *word* do not specify a file descriptor open for output. If *word* evaluates to **–**, file descriptor *n* is closed. As a special case, if *n* is omitted, and *word* does not expand to one or more digits or **–**, this redirects the standard output and standard error as described previously.

Moving File Descriptors

The redirection operator

[*n*]**<&***digit***–**

moves the file descriptor *digit* to file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified. *digit* is closed after being duplicated to *n*.

Similarly, the redirection operator

[*n*]**>&***digit***–**

moves the file descriptor *digit* to file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified.

Opening File Descriptors for Reading and Writing

The redirection operator

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

opens the file whose name is the expansion of *word* for both reading and writing on file descriptor *n*, or on file descriptor 0 if *n* is not specified. If the file does not exist, it is created.

ALIASES

Aliases allow a string to be substituted for a word that is in a position in the input where it can be the first word of a simple command. Aliases have names and corresponding values that are set and unset using the **alias** and **unalias** builtin commands (see **SHELL BUILTIN COMMANDS** below).

If the shell reads an unquoted word in the right position, it checks the word to see if it matches an alias name. If it matches, the shell replaces the word with the alias value, and reads that value as if it had been read instead of the word. The shell doesn't look at any characters following the word before attempting alias substitution.

The characters **/**, **$**, **`**, and **=** and any of the shell *metacharacters* or quoting characters listed above may not appear in an alias name. The replacement text may contain any valid shell input, including shell metacharacters. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias **ls** to **ls –F**, for instance, and **bash** does not try to recursively expand the replacement text.

If the last character of the alias value is a *blank*, the shell checks the next command word following the alias for alias expansion.

Aliases are created and listed with the **alias** command, and removed with the **unalias** command.

There is no mechanism for using arguments in the replacement text. If arguments are needed, use a shell function (see **FUNCTIONS** below) instead.

Aliases are not expanded when the shell is not interactive, unless the **expand_aliases** shell option is set using **shopt** (see the description of **shopt** under **SHELL BUILTIN COMMANDS** below).

The rules concerning the definition and use of aliases are somewhat confusing. **Bash** always reads at least one complete line of input, and all lines that make up a compound command, before executing any of the commands on that line or the compound command. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the shell reads the next line of input, and an alias definition in a compound command does not take effect until the shell parses and executes the entire compound command. The commands following the alias definition on that line, or in the rest of a compound command, are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when a function definition is read, not when the function is executed, because a function definition is itself a command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use **alias** in compound commands.

For almost every purpose, shell functions are preferable to aliases.

FUNCTIONS

A shell function, defined as described above under **SHELL GRAMMAR**, stores a series of commands for later execution. When the name of a shell function is used as a simple command name, the shell executes the list of commands associated with that function name. Functions are executed in the context of the calling shell; there is no new process created to interpret them (contrast this with the execution of a shell script).

When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter **#** is updated to reflect the new positional parameters. Special parameter **0** is unchanged. The first element of the **FUNCNAME** variable is set to the name of the function while the function is executing.

All other aspects of the shell execution environment are identical between a function and its caller with these exceptions: the **DEBUG** and **RETURN** traps (see the description of the **trap** builtin under **SHELL BUILTIN COMMANDS** below) are not inherited unless the function has been given the **trace** attribute (see the description of the **declare** builtin below) or the **−o functrace** shell option has been enabled with the **set** builtin (in which case all functions inherit the **DEBUG** and **RETURN** traps), and the **ERR** trap is not inherited unless the **−o errtrace** shell option has been enabled.

Variables local to the function are declared with the **local** builtin command (*local variables*). Ordinarily, variables and their values are shared between the function and its caller. If a variable is declared **local**, the variable's visible scope is restricted to that function and its children (including the functions it calls).

In the following description, the *current scope* is a currently- executing function. Previous scopes consist of that function's caller and so on, back to the "global" scope, where the shell is not executing any shell function. A local variable at the current scope is a variable declared using the **local** or **declare** builtins in the function that is currently executing.

Local variables "shadow" variables with the same name declared at previous scopes. For instance, a local variable declared in a function hides variables with the same name declared at previous scopes, including global variables: references and assignments refer to the local variable, leaving the variables at previous scopes unmodified. When the function returns, the global variable is once again visible.

The shell uses *dynamic scoping* to control a variable's visibility within functions. With dynamic scoping, visible variables and their values are a result of the sequence of function calls that caused execution to reach the current function. The value of a variable that a function sees depends on its value within its caller, if any, whether that caller is the global scope or another shell function. This is also the value that a local variable declaration shadows, and the value that is restored when the function returns.

For example, if a variable *var* is declared as local in function *func1*, and *func1* calls another function *func2*, references to *var* made from within *func2* resolve to the local variable *var* from *func1*, shadowing any global variable named *var*.

The **unset** builtin also acts using the same dynamic scope: if a variable is local to the current scope, **unset** unsets it; otherwise the unset will refer to the variable found in any calling scope as described above. If a variable at the current local scope is unset, it remains so (appearing as unset) until it is reset in that scope or until the function returns. Once the function returns, any instance of the variable at a previous scope becomes visible. If the unset acts on a variable at a previous scope, any instance of a variable with that name that had been shadowed becomes visible (see below how the **localvar_unset** shell option changes this behavior).

The **FUNCNEST** variable, if set to a numeric value greater than 0, defines a maximum function nesting level. Function invocations that exceed the limit cause the entire command to abort.

If the builtin command **return** is executed in a function, the function completes and execution resumes with the next command after the function call. If **return** is supplied a numeric argument, that is the function's return status; otherwise the function's return status is the exit status of the last command executed before the **return**. Any command associated with the **RETURN** trap is executed before execution resumes. When a function completes, the values of the positional parameters and the special parameter **#** are restored to the values they had prior to the function's execution.

The **−f** option to the **declare** or **typeset** builtin commands lists function names and definitions. The **−F** option to **declare** or **typeset** lists the function names only (and optionally the source file and line number, if the **extdebug** shell option is enabled). Functions may be exported so that child shell processes (those created when executing a separate shell invocation) automatically have them defined with the **−f** option to the **export** builtin. The **−f** option to the **unset** builtin deletes a function definition.

Functions may be recursive. The **FUNCNEST** variable may be used to limit the depth of the function call stack and restrict the number of function invocations. By default, **bash** imposes no limit on the number of recursive calls.

ARITHMETIC EVALUATION

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the **let** and **declare** builtin commands, the **((** compound command, the arithmetic **for** command, the **[[** conditional command, and **Arithmetic Expansion**).

Evaluation is done in the largest fixed-width integers available, with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

id++ *id*−−
: variable post-increment and post-decrement

++*id* −−*id*
: variable pre-increment and pre-decrement

− + unary minus and plus

! ~ logical and bitwise negation

****** exponentiation

*** / %** multiplication, division, remainder

+ − addition, subtraction

<< >> left and right bitwise shifts

<= >= < >
: comparison

== != equality and inequality

& bitwise AND

^ bitwise exclusive OR

| bitwise OR

&& logical AND

|| logical OR

*expr***?***expr***:***expr*
conditional operator

= *= /= %= += −= <<= >>= &= ^= |=
assignment

expr1 **,** *expr2*
comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. This means you can use "x", where *x* is a shell variable name, in an arithmetic expression, and the shell will evaluate its value as an expression and use the result. A shell variable that is null or unset evaluates to 0 when referenced by name in an expression.

The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the *integer* attribute using **declare −i** is assigned a value. A null value evaluates to 0. A shell variable need not have its *integer* attribute turned on to be used in an expression.

Integer constants follow the C language definition, without suffixes or character constants. Constants with a leading 0 are interpreted as octal numbers. A leading 0x or 0X denotes hexadecimal. Otherwise, numbers take the form [*base#*]n, where the optional *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base#* is omitted, then base 10 is used. When specifying *n*, if a non-digit is required, the digits greater than 9 are represented by the lowercase letters, the uppercase letters, @, and _, in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in precedence order. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

CONDITIONAL EXPRESSIONS

Conditional expressions are used by the **[[** compound command and the **test** and **[** builtin commands to test file attributes and perform string and arithmetic comparisons. The **test** and **[** commands determine their behavior based on the number of arguments; see the descriptions of those commands for any other command-specific actions.

Expressions are formed from the unary or binary primaries listed below. Unary expressions are often used to examine the status of a file or shell variable. Binary operators are used for string, numeric, and file attribute comparisons.

Bash handles several filenames specially when they are used in expressions. If the operating system on which **bash** is running provides these special files, bash will use them; otherwise it will emulate them internally with this behavior: If any *file* argument to one of the primaries is of the form */dev/fd/n*, then **bash** checks file descriptor *n*. If the *file* argument to one of the primaries is one of */dev/stdin*, */dev/stdout*, or */dev/stderr*, **bash** checks file descriptor 0, 1, or 2, respectively.

Unless otherwise specified, primaries that operate on files follow symbolic links and operate on the target of the link, rather than the link itself.

When used with **[[**, or when the shell is in posix mode, the **<** and **>** operators sort lexicographically using the current locale. When the shell is not in posix mode, the **test** command sorts using ASCII ordering.

−a *file* True if *file* exists.

−b *file* True if *file* exists and is a block special file.

−c *file* True if *file* exists and is a character special file.

−d *file* True if *file* exists and is a directory.

−e *file* True if *file* exists.

−f *file* True if *file* exists and is a regular file.

−g *file* True if *file* exists and is set-group-id.
−h *file* True if *file* exists and is a symbolic link.
−k *file* True if *file* exists and its “sticky” bit is set.
−p *file* True if *file* exists and is a named pipe (FIFO).
−r *file* True if *file* exists and is readable.
−s *file* True if *file* exists and has a size greater than zero.
−t *fd* True if file descriptor *fd* is open and refers to a terminal.
−u *file* True if *file* exists and its set-user-id bit is set.
−w *file* True if *file* exists and is writable.
−x *file* True if *file* exists and is executable.
−G *file* True if *file* exists and is owned by the effective group id.
−L *file* True if *file* exists and is a symbolic link.
−N *file* True if *file* exists and has been modified since it was last accessed.
−O *file* True if *file* exists and is owned by the effective user id.
−S *file* True if *file* exists and is a socket.

−o *optname*
True if the shell option *optname* is enabled. See the list of options under the description of the **−o** option to the **set** builtin below.

−v *varname*
True if the shell variable *varname* is set (has been assigned a value). If *varname* is an indexed array variable name subscripted by @ or *, this returns true if the array has any set elements. If *varname* is an associative array variable name subscripted by @ or *, this returns true if an element with that key is set.

−R *varname*
True if the shell variable *varname* is set and is a name reference.

−z *string*
True if the length of *string* is zero.

string
−n *string*
True if the length of *string* is non-zero.

string1 **==** *string2*
string1 **=** *string2*
True if the strings are equal. **=** should be used with the **test** command for POSIX conformance. When used with the **[[** command, this performs pattern matching as described above (**Compound Commands**).

string1 **!=** *string2*
True if the strings are not equal.

string1 **<** *string2*
True if *string1* sorts before *string2* lexicographically.

string1 **>** *string2*
True if *string1* sorts after *string2* lexicographically.

file1 **−ef** *file2*
True if *file1* and *file2* refer to the same device and inode numbers.

file1 **−nt** *file2*
True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.

file1 **−ot** *file2*
True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

arg1 **OP** *arg2*
OP is one of **−eq**, **−ne**, **−lt**, **−le**, **−gt**, or **−ge**. These arithmetic binary operators return true if *arg1* is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to *arg2*, respectively. *arg1* and *arg2* may be positive or negative integers. When used with the **[[** command, *arg1* and *arg2* are evaluated as arithmetic expressions (see **ARITHMETIC EVALUATION** above). Since the expansions the **[[** command performs on *arg1* and *arg2* can potentially

result in empty strings, arithmetic expression evaluation treats those as expressions that evaluate to 0.

SIMPLE COMMAND EXPANSION

When the shell executes a simple command, it performs the following expansions, assignments, and redirections, from left to right, in the following order.

1. The words that the parser has marked as variable assignments (those preceding the command name) and redirections are saved for later processing.
2. The words that are not variable assignments or redirections are expanded. If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.
3. Redirections are performed as described above under **REDIRECTION**.
4. The text after the **=** in each variable assignment undergoes tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal before being assigned to the variable.

If no command name results, the variable assignments affect the current shell environment. In the case of such a command (one that consists only of assignment statements and redirections), assignment statements are performed before redirections. Otherwise, the variables are added to the environment of the executed command and do not affect the current shell environment. If any of the assignments attempts to assign a value to a readonly variable, an error occurs, and the command exits with a non-zero status.

If no command name results, redirections are performed, but do not affect the current shell environment. A redirection error causes the command to exit with a non-zero status.

If there is a command name left after expansion, execution proceeds as described below. Otherwise, the command exits. If one of the expansions contained a command substitution, the exit status of the command is the exit status of the last command substitution performed. If there were no command substitutions, the command exits with a zero status.

COMMAND EXECUTION

After a command has been split into words, if it results in a simple command and an optional list of arguments, the shell performs the following actions.

If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described above in **FUNCTIONS**. If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.

If the name is neither a shell function nor a builtin, and contains no slashes, **bash** searches each element of the **PATH** for a directory containing an executable file by that name. **Bash** uses a hash table to remember the full pathnames of executable files (see **hash** under **SHELL BUILTIN COMMANDS** below). Bash performs a full search of the directories in **PATH** only if the command is not found in the hash table. If the search is unsuccessful, the shell searches for a defined shell function named **command_not_found_handle**. If that function exists, it is invoked in a separate execution environment with the original command and the original command's arguments as its arguments, and the function's exit status becomes the exit status of that subshell. If that function is not defined, the shell prints an error message and returns an exit status of 127.

If the search is successful, or if the command name contains one or more slashes, the shell executes the named program in a separate execution environment. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments given, if any.

If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a *shell script*, a file containing shell commands, and the shell creates a new instance of itself to execute it. Bash tries to determine whether the file is a text file or a binary, and will not execute files it determines to be binaries. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to handle the script, with the exception that the locations of commands remembered by the parent (see **hash** below under **SHELL BUILTIN COMMANDS** are retained by the child.

If the program is a file beginning with **#!**, the remainder of the first line specifies an interpreter for the program. The shell executes the specified interpreter on operating systems that do not handle this executable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any.

COMMAND EXECUTION ENVIRONMENT

The shell has an *execution environment*, which consists of the following:

- Open files inherited by the shell at invocation, as modified by redirections supplied to the **exec** builtin.
- The current working directory as set by **cd**, **pushd**, or **popd**, or inherited by the shell at invocation.
- The file creation mode mask as set by **umask** or inherited from the shell's parent.
- Current traps set by **trap**.
- Shell parameters that are set by variable assignment or with **set** or inherited from the shell's parent in the environment.
- Shell functions defined during execution or inherited from the shell's parent in the environment.
- Options enabled at invocation (either by default or with command-line arguments) or by **set**.
- Options enabled by **shopt**.
- Shell aliases defined with **alias**.
- Various process IDs, including those of background jobs, the value of **$$**, and the value of **PPID**.

When a simple command other than a builtin or shell function is to be executed, it is invoked in a separate execution environment that consists of the following. Unless otherwise noted, the values are inherited from the shell.

- The shell's open files, plus any modifications and additions specified by redirections to the command.
- The current working directory.
- The file creation mode mask.
- Shell variables and functions marked for export, along with variables exported for the command, passed in the environment.
- Traps caught by the shell are reset to the values inherited from the shell's parent, and traps ignored by the shell are ignored.

A command invoked in this separate environment cannot affect the shell's execution environment.

A *subshell* is a copy of the shell process.

Command substitution, commands grouped with parentheses, and asynchronous commands are invoked in a subshell environment that is a duplicate of the shell environment, except that traps caught by the shell are reset to the values that the shell inherited from its parent at invocation. Builtin commands that are invoked as part of a pipeline, except possibly in the last element depending on the value of the **lastpipe** shell option, are also executed in a subshell environment. Changes made to the subshell environment cannot affect the shell's execution environment.

When the shell is in posix mode, subshells spawned to execute command substitutions inherit the value of the **–e** option from their parent shell. When not in posix mode, **bash** clears the **–e** option in such subshells. See the description of the **inherit_errexit** shell option below for how to control this behavior when not in posix mode.

If a command is followed by a **&** and job control is not active, the default standard input for the command is the empty file */dev/null*. Otherwise, the invoked command inherits the file descriptors of the calling shell as modified by redirections.

ENVIRONMENT

When a program is invoked it is given an array of strings called the *environment*. This is a list of *name*−*value* pairs, of the form *name*=*value*.

The shell provides several ways to manipulate the environment. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for *export* to child processes. Executed commands inherit the environment. The **export**, **declare −x**, and **unset** commands modify the environment by adding and deleting parameters and functions. If the value of a parameter in the environment is modified, the new value automatically becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the **unset** or **export −n** commands, plus any additions via the **export** and **declare −x** commands.

If any parameter assignments, as described above in **PARAMETERS**, appear before a *simple command*, the variable assignments are part of that command's environment for as long as it executes. These assignment statements affect only the environment seen by that command. If these assignments precede a call to a shell function, the variables are local to the function and exported to that function's children.

If the **−k** option is set (see the **set** builtin command below), then *all* parameter assignments are placed in the environment for a command, not just those that precede the command name.

When **bash** invokes an external command, the variable **_** is set to the full pathname of the command and passed to that command in its environment.

EXIT STATUS

The exit status of an executed command is the value returned by the *waitpid* system call or equivalent function. Exit statuses fall between 0 and 255, though, as explained below, the shell may use values above 125 specially. Exit statuses from shell builtins and compound commands are also limited to this range. Under certain circumstances, the shell will use special values to indicate specific failure modes.

For the shell's purposes, a command which exits with a zero exit status has succeeded. So while an exit status of zero indicates success, a non-zero exit status indicates failure.

When a command terminates on a fatal signal *N*, **bash** uses the value of 128+*N* as the exit status.

If a command is not found, the child process created to execute it returns a status of 127. If a command is found but is not executable, the return status is 126.

If a command fails because of an error during expansion or redirection, the exit status is greater than zero.

Shell builtin commands return a status of 0 (*true*) if successful, and non-zero (*false*) if an error occurs while they execute. All builtins return an exit status of 2 to indicate incorrect usage, generally invalid options or missing arguments.

The exit status of the last command is available in the special parameter $?.

Bash itself returns the exit status of the last command executed, unless a syntax error occurs, in which case it exits with a non-zero value. See also the **exit** builtin command below.

SIGNALS

When **bash** is interactive, in the absence of any traps, it ignores **SIGTERM** (so that **kill 0** does not kill an interactive shell), and catches and handles **SIGINT** (so that the **wait** builtin is interruptible). When **bash** receives **SIGINT**, it breaks out of any executing loops. In all cases, **bash** ignores **SIGQUIT**. If job control is in effect, **bash** ignores **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

The **trap** builtin modifies the shell's signal handling, as described below.

Non-builtin commands **bash** executes have signal handlers set to the values inherited by the shell from its parent, unless **trap** sets them to be ignored, in which case the child process will ignore them as well. When job control is not in effect, asynchronous commands ignore **SIGINT** and **SIGQUIT** in addition to these inherited handlers. Commands run as a result of command substitution ignore the keyboard-generated job control signals **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

The shell exits by default upon receipt of a **SIGHUP**. Before exiting, an interactive shell resends the

SIGHUP to all jobs, running or stopped. The shell sends **SIGCONT** to stopped jobs to ensure that they receive the **SIGHUP** (see **JOB CONTROL** below for more information about running and stopped jobs). To prevent the shell from sending the signal to a particular job, remove it from the jobs table with the **disown** builtin (see **SHELL BUILTIN COMMANDS** below) or mark it not to receive **SIGHUP** using **disown −h**.

If the **huponexit** shell option has been set using **shopt**, **bash** sends a **SIGHUP** to all jobs when an interactive login shell exits.

If **bash** is waiting for a command to complete and receives a signal for which a trap has been set, it will not execute the trap until the command completes. If **bash** is waiting for an asynchronous command via the **wait** builtin, and it receives a signal for which a trap has been set, the **wait** builtin will return immediately with an exit status greater than 128, immediately after which the shell executes the trap.

When job control is not enabled, and **bash** is waiting for a foreground command to complete, the shell receives keyboard-generated signals such as **SIGINT** (usually generated by **^C**) that users commonly intend to send to that command. This happens because the shell and the command are in the same process group as the terminal, and **^C** sends **SIGINT** to all processes in that process group. Since **bash** does not enable job control by default when the shell is not interactive, this scenario is most common in non-interactive shells.

When job control is enabled, and **bash** is waiting for a foreground command to complete, the shell does not receive keyboard-generated signals, because it is not in the same process group as the terminal. This scenario is most common in interactive shells, where **bash** attempts to enable job control by default. See **JOB CONTROL** below for more information about process groups.

When job control is not enabled, and **bash** receives **SIGINT** while waiting for a foreground command, it waits until that foreground command terminates and then decides what to do about the **SIGINT**:

1. If the command terminates due to the **SIGINT**, **bash** concludes that the user meant to send the **SIGINT** to the shell as well, and acts on the **SIGINT** (e.g., by running a **SIGINT** trap, exiting a non-interactive shell, or returning to the top level to read a new command).
2. If the command does not terminate due to **SIGINT**, the program handled the **SIGINT** itself and did not treat it as a fatal signal. In that case, **bash** does not treat **SIGINT** as a fatal signal, either, instead assuming that the **SIGINT** was used as part of the program's normal operation (e.g., emacs uses it to abort editing commands) or deliberately discarded. However, **bash** will run any trap set on **SIGINT**, as it does with any other trapped signal it receives while it is waiting for the foreground command to complete, for compatibility.

When job control is enabled, **bash** does not receive keyboard-generated signals such as **SIGINT** while it is waiting for a foreground command. An interactive shell does not pay attention to the **SIGINT**, even if the foreground command terminates as a result, other than noting its exit status. If the shell is not interactive, and the foreground command terminates due to the **SIGINT**, **bash** pretends it received the **SIGINT** itself (scenario 1 above), for compatibility.

JOB CONTROL

Job control refers to the ability to selectively stop (*suspend*) the execution of processes and continue (*resume*) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the operating system kernel's terminal driver and **bash**.

The shell associates a *job* with each pipeline. It keeps a table of currently executing jobs, which the **jobs** command will display. Each job has a *job number*, which **jobs** displays between brackets. Job numbers start at 1. When **bash** starts a job asynchronously (in the *background*), it prints a line that looks like:

    [1] 25647

indicating that this job is job number 1 and that the process ID of the last process in the pipeline associated with this job is 25647. All of the processes in a single pipeline are members of the same job. **Bash** uses the *job* abstraction as the basis for job control.

To facilitate the implementation of the user interface to job control, each process has a *process group ID*, and the operating system maintains the notion of a *current terminal process group ID*. This terminal process group ID is associated with the *controlling terminal*.

Processes that have the same process group ID are said to be part of the same *process group*. Members of the *foreground* process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as **SIGINT**. Processes in the foreground process group are said to be *foreground* processes. *Background* processes are those whose process group ID differs from the controlling terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or, if the user so specifies with "stty tostop", write to the controlling terminal. The system sends a **SIGTTIN (SIGTTOU)** signal to background processes which attempt to read from (write to when "tostop" is in effect) the terminal, which, unless caught, suspends the process.

If the operating system on which **bash** is running supports job control, **bash** contains facilities to use it. Typing the *suspend* character (typically **^Z**, Control-Z) while a process is running stops that process and returns control to **bash**. Typing the *delayed suspend* character (typically **^Y**, Control-Y) causes the process stop when it attempts to read input from the terminal, and returns control to **bash**. The user then manipulates the state of this job, using the **bg** command to continue it in the background, the **fg** command to continue it in the foreground, or the **kill** command to kill it. The suspend character takes effect immediately, and has the additional side effect of discarding any pending output and typeahead. To force a background process to stop, or stop a process that's not associated with the current terminal session, send it the **SIGSTOP** signal using **kill**.

There are a number of ways to refer to a job in the shell. The **%** character introduces a job specification (jobspec).

Job number *n* may be referred to as **%n**. A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, **%ce** refers to a job whose command name begins with **ce**. Using **%?ce**, on the other hand, refers to any job containing the string **ce** in its command line. If the prefix or substring matches more than one job, **bash** reports an error.

The symbols **%%** and **%+** refer to the shell's notion of the *current job*. A single % (with no accompanying job specification) also refers to the current job. **%−** refers to the *previous job*. When a job starts in the background, a job stops while in the foreground, or a job is resumed in the background, it becomes the current job. The job that was the current job becomes the previous job. When the current job terminates, the previous job becomes the current job. If there is only a single job, **%+** and **%−** can both be used to refer to that job. In output pertaining to jobs (e.g., the output of the **jobs** command), the current job is always marked with a **+**, and the previous job with a **−**.

Simply naming a job can be used to bring it into the foreground: **%1** is a synonym for "fg %1", bringing job 1 from the background into the foreground. Similarly, "%1 &" resumes job 1 in the background, equivalent to "bg %1".

The shell learns immediately whenever a job changes state. Normally, **bash** waits until it is about to print a prompt before notifying the user about changes in a job's status so as to not interrupt any other output, though it will notify of changes in a job's status after a foreground command in a list completes, before executing the next command in the list. If the **−b** option to the **set** builtin command is enabled, **bash** reports status changes immediately. **Bash** executes any trap on **SIGCHLD** for each child that terminates.

When a job terminates and **bash** notifies the user about it, **bash** removes the job from the table. It will not appear in **jobs** output, but **wait** will report its exit status, as long as it's supplied the process ID associated with the job as an argument. When the table is empty, job numbers start over at 1.

If a user attempts to exit **bash** while jobs are stopped (or, if the **checkjobs** shell option has been enabled using the **shopt** builtin, running), the shell prints a warning message, and, if the **checkjobs** option is enabled, lists the jobs and their statuses. The **jobs** command may then be used to inspect their status. If the user immediately attempts to exit again, without an intervening command, **bash** does not print another warning, and terminates any stopped jobs.

When the shell is waiting for a job or process using the **wait** builtin, and job control is enabled, **wait** will return when the job changes state. The **−f** option causes **wait** to wait until the job or process terminates before returning.

PROMPTING

When executing interactively, **bash** displays the primary prompt **PS1** when it is ready to read a command, and the secondary prompt **PS2** when it needs more input to complete a command.

Bash examines the value of the array variable **PROMPT_COMMAND** just before printing each primary prompt. If any elements in **PROMPT_COMMAND** are set and non-null, Bash executes each value, in numeric order, just as if it had been typed on the command line. **Bash** displays **PS0** after it reads a command but before executing it.

Bash displays **PS4** as described above before tracing each command when the **−x** option is enabled.

Bash allows the prompt strings **PS0**, **PS1**, **PS2**, and **PS4**, to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

- **\a** An ASCII bell character (07).
- **\d** The date in "Weekday Month Date" format (e.g., "Tue May 26").
- **\D{***format***}**
  The *format* is passed to *strftime*(3) and the result is inserted into the prompt string; an empty *format* results in a locale-specific time representation. The braces are required.
- **\e** An ASCII escape character (033).
- **\h** The hostname up to the first ".".
- **\H** The hostname.
- **\j** The number of jobs currently managed by the shell.
- **\l** The basename of the shell's terminal device name (e.g., "ttys0").
- **\n** A newline.
- **\r** A carriage return.
- **\s** The name of the shell: the basename of **$0** (the portion following the final slash).
- **\t** The current time in 24-hour HH:MM:SS format.
- **\T** The current time in 12-hour HH:MM:SS format.
- **\@** The current time in 12-hour am/pm format.
- **\A** The current time in 24-hour HH:MM format.
- **\u** The username of the current user.
- **\v** The **bash** version (e.g., 2.00).
- **\V** The **bash** release, version + patch level (e.g., 2.00.0)
- **\w** The value of the **PWD** shell variable (**$PWD**), with **$HOME** abbreviated with a tilde (uses the value of the **PROMPT_DIRTRIM** variable).
- **\W** The basename of **$PWD**, with **$HOME** abbreviated with a tilde.
- **\!** The history number of this command.
- **\#** The command number of this command.
- **\$** If the effective UID is 0, a **#**, otherwise a **$**.
- *****nnn* The character corresponding to the octal number *nnn*.
- **** A backslash.
- **\[** Begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt.
- **\]** End a sequence of non-printing characters.

The command number and the history number are usually different: the history number of a command is its position in the history list, which may include commands restored from the history file (see **HISTORY** below), while the command number is the position in the sequence of commands executed during the current shell session. After the string is decoded, it is expanded via parameter expansion, command substitution, arithmetic expansion, and quote removal, subject to the value of the **promptvars** shell option (see the description of the **shopt** command under **SHELL BUILTIN COMMANDS** below). This can have unwanted side effects if escaped portions of the string appear within command substitution or contain characters special to word expansion.

READLINE

This is the library that handles reading input when using an interactive shell, unless the **−−noediting** option is supplied at shell invocation. Line editing is also used when using the **−e** option to the **read** builtin. By default, the line editing commands are similar to those of emacs; a vi-style line editing interface is also available. Line editing can be enabled at any time using the **−o emacs** or **−o vi** options to the **set** builtin (see **SHELL BUILTIN COMMANDS** below). To turn off line editing after the shell is running, use the **+o emacs** or **+o vi** options to the **set** builtin.

Readline Notation

This section uses Emacs-style editing concepts and uses its notation for keystrokes. Control keys are denoted by C−*key*, e.g., C−n means Control−N. Similarly, *meta* keys are denoted by M−*key*, so M−x means Meta−X. The Meta key is often labeled “Alt” or “Option”.

On keyboards without a *Meta* key, M−*x* means ESC *x*, i.e., press and release the Escape key, then press and release the *x* key, in sequence. This makes ESC the *meta prefix*. The combination M−C−*x* means ESC Control−*x*: press and release the Escape key, then press and hold the Control key while pressing the *x* key, then release both.

On some keyboards, the Meta key modifier produces characters with the eighth bit (0200) set. You can use the **enable−meta−key** variable to control whether or not it does this, if the keyboard allows it. On many others, the terminal or terminal emulator converts the metafied key to a key sequence beginning with ESC as described in the preceding paragraph.

If your *Meta* key produces a key sequence with the ESC meta prefix, you can make M-*key* key bindings you specify (see **Readline Key Bindings** below) do the same thing by setting the **force−meta−prefix** variable.

Readline commands may be given numeric *arguments*, which normally act as a repeat count. Sometimes, however, it is the sign of the argument that is significant. Passing a negative argument to a command that acts in the forward direction (e.g., **kill−line**) makes that command act in a backward direction. Commands whose behavior with arguments deviates from this are noted below.

The *point* is the current cursor position, and *mark* refers to a saved cursor position. The text between the point and mark is referred to as the *region*. **Readline** has the concept of an *active region*: when the region is active, **readline** redisplay highlights the region using the value of the **active-region-start-color** variable. The **enable−active−region** variable turns this on and off. Several commands set the region to active; those are noted below.

When a command is described as *killing* text, the text deleted is saved for possible future retrieval (*yanking*). The killed text is saved in a *kill ring*. Consecutive kills accumulate the deleted text into one unit, which can be yanked all at once. Commands which do not kill text separate the chunks of text on the kill ring.

Readline Initialization

Readline is customized by putting commands in an initialization file (the *inputrc* file). The name of this file is taken from the value of the **INPUTRC** shell variable. If that variable is unset, the default is *~/.inputrc*. If that file does not exist or cannot be read, **readline** looks for */etc/inputrc*. When a program that uses the **readline** library starts up, **readline** reads the initialization file and sets the key bindings and variables found there, before reading any user input.

There are only a few basic constructs allowed in the inputrc file. Blank lines are ignored. Lines beginning with a **#** are comments. Lines beginning with a **$** indicate conditional constructs. Other lines denote key bindings and variable settings.

The default key-bindings in this section may be changed using key binding commands in the *inputrc* file. Programs that use the **readline** library, including **bash**, may add their own commands and bindings.

For example, placing

    M−Control−u: universal−argument

or

    C−Meta−u: universal−argument

into the *inputrc* would make M−C−u execute the **readline** command *universal−argument*.

Key bindings may contain the following symbolic character names: *DEL*, *ESC*, *ESCAPE*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *RUBOUT* (a destructive backspace), *SPACE*, *SPC*, and *TAB*.

In addition to command names, **readline** allows keys to be bound to a string that is inserted when the key is pressed (a *macro*). The difference between a macro and a command is that a macro is enclosed in single or double quotes.

Readline Key Bindings

The syntax for controlling key bindings in the *inputrc* file is simple. All that is required is the name of the command or the text of a macro and a key sequence to which it should be bound. The key sequence may be specified in one of two ways: as a symbolic key name, possibly with *Meta−* or *Control−* prefixes, or as a key sequence composed of one or more characters enclosed in double quotes. The key sequence and name are separated by a colon. There can be no whitespace between the name and the colon.

When using the form **keyname**:*function−name* or *macro*, *keyname* is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "> output"
```

In the above example, *C−u* is bound to the function **universal−argument**, *M−DEL* is bound to the function **backward−kill−word**, and *C−o* is bound to run the macro expressed on the right hand side (that is, to insert the text “> output” into the line).

In the second form, **"keyseq"**:*function−name* or *macro*, **keyseq** differs from **keyname** above in that strings denoting an entire key sequence may be specified by placing the sequence within double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but none of the symbolic character names are recognized.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In this example, *C−u* is again bound to the function **universal−argument**. *C−x C−r* is bound to the function **re−read−init−file**, and *ESC [1 1 ~* is bound to insert the text “Function Key 1”.

The full set of GNU Emacs style escape sequences available when specifying key sequences is

- **\C−** A control prefix.
- **\M−** Adding the meta prefix or converting the following character to a meta character, as described below under **force-meta-prefix**.
- **\e** An escape character.
- **** Backslash.
- **\"** Literal ", a double quote.
- **\'** Literal ', a single quote.

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

- **\a** alert (bell)
- **\b** backspace
- **\d** delete
- **\f** form feed
- **\n** newline
- **\r** carriage return
- **\t** horizontal tab
- **\v** vertical tab
- *****nnn* The eight-bit character whose value is the octal value *nnn* (one to three digits).
- **\x***HH* The eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits).

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. The backslash escapes described above are expanded in the macro body. Backslash quotes any other character in the macro text, including " and '.

Bash will display or modify the current **readline** key bindings with the **bind** builtin command. The **−o emacs** or **−o vi** options to the **set** builtin (see **SHELL BUILTIN COMMANDS** below) change the editing mode during interactive use.

Readline Variables

Readline has variables that can be used to further customize its behavior. A variable may be set in the *inputrc* file with a statement of the form

> **set** *variable−name value*

or using the **bind** builtin command (see **SHELL BUILTIN COMMANDS** below).

Except where noted, **readline** variables can take the values **On** or **Off** (without regard to case). Unrecognized variable names are ignored. When **readline** reads a variable value, empty or null values, "on" (case-insensitive), and "1" are equivalent to **On**. All other values are equivalent to **Off**.

The **bind −V** command lists the current **readline** variable names and values (see **SHELL BUILTIN COMMANDS** below).

The variables and their default values are:

active−region−start−color
: A string variable that controls the text color and background when displaying the text in the active region (see the description of **enable−active−region** below). This string must not take up any physical character positions on the display, so it should consist only of terminal escape sequences. It is output to the terminal before displaying the text in the active region. This variable is reset to the default value whenever the terminal type changes. The default value is the string that puts the terminal in standout mode, as obtained from the terminal's terminfo description. A sample value might be "\e[01;33m".

active−region−end−color
: A string variable that "undoes" the effects of **active−region−start−color** and restores "normal" terminal display appearance after displaying text in the active region. This string must not take up any physical character positions on the display, so it should consist only of terminal escape sequences. It is output to the terminal after displaying the text in the active region. This variable is reset to the default value whenever the terminal type changes. The default value is the string that restores the terminal from standout mode, as obtained from the terminal's terminfo description. A sample value might be "\e[0m".

bell−style (audible)
: Controls what happens when **readline** wants to ring the terminal bell. If set to **none**, **readline** never rings the bell. If set to **visible**, **readline** uses a visible bell if one is available. If set to **audible**, **readline** attempts to ring the terminal's bell.

bind−tty−special−chars (On)
: If set to **On**, **readline** attempts to bind the control characters that are treated specially by the kernel's terminal driver to their **readline** equivalents. These override the default **readline** bindings described here. Type "stty −a" at a **bash** prompt to see your current terminal settings, including the special control characters (usually **cchars**).

blink−matching−paren (Off)
: If set to **On**, **readline** attempts to briefly move the cursor to an opening parenthesis when a closing parenthesis is inserted.

colored−completion−prefix (Off)
: If set to **On**, when listing completions, **readline** displays the common prefix of the set of possible completions using a different color. The color definitions are taken from the value of the **LS_COLORS** environment variable. If there is a color definition in **$LS_COLORS** for the custom suffix ".readline-colored-completion-prefix", **readline** uses this color for the common prefix instead of its default.

colored−stats (Off)
: If set to **On**, **readline** displays possible completions using different colors to indicate their file type. The color definitions are taken from the value of the **LS_COLORS** environment variable.

comment−begin ("**#**")
: The string that the **readline insert−comment** command inserts. This command is bound to **M−#** in emacs mode and to **#** in vi command mode.

completion−display−width (−1)
The number of screen columns used to display possible matches when performing completion. The value is ignored if it is less than 0 or greater than the terminal screen width. A value of 0 causes matches to be displayed one per line. The default value is −1.

completion−ignore−case (Off)
If set to **On**, **readline** performs filename matching and completion in a case−insensitive fashion.

completion−map−case (Off)
If set to **On**, and **completion−ignore−case** is enabled, **readline** treats hyphens (*−*) and underscores (*_*) as equivalent when performing case−insensitive filename matching and completion.

completion−prefix−display−length (0)
The maximum length in characters of the common prefix of a list of possible completions that is displayed without modification. When set to a value greater than zero, **readline** replaces common prefixes longer than this value with an ellipsis when displaying possible completions. If a completion begins with a period, and **eadline** is completing filenames, it uses three underscores instead of an ellipsis.

completion−query−items (100)
This determines when the user is queried about viewing the number of possible completions generated by the **possible−completions** command. It may be set to any integer value greater than or equal to zero. If the number of possible completions is greater than or equal to the value of this variable, **readline** asks whether or not the user wishes to view them; otherwise **readline** simply lists them on the terminal. A zero value means **readline** should never ask; negative values are treated as zero.

convert−meta (On)
If set to **On**, **readline** converts characters it reads that have the eighth bit set to an ASCII key sequence by clearing the eighth bit and prefixing it with an escape character (converting the character to have the meta prefix). The default is *On*, but **readline** sets it to *Off* if the locale contains characters whose encodings may include bytes with the eighth bit set. This variable is dependent on the **LC_CTYPE** locale category, and may change if the locale changes. This variable also affects key bindings; see the description of **force−meta−prefix** below.

disable−completion (Off)
If set to **On**, **readline** inhibits word completion. Completion characters are inserted into the line as if they had been mapped to **self-insert**.

echo−control−characters (On)
When set to **On**, on operating systems that indicate they support it, **readline** echoes a character corresponding to a signal generated from the keyboard.

editing−mode (emacs)
Controls whether **readline** uses a set of key bindings similar to *Emacs* or *vi*. **editing−mode** can be set to either **emacs** or **vi**.

emacs−mode−string (@)
If the *show−mode−in−prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when emacs editing mode is active. The value is expanded like a key binding, so the standard set of meta- and control- prefixes and backslash escape sequences is available. The \1 and \2 escapes begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string.

enable−active−region (On)
When this variable is set to *On*, **readline** allows certain commands to designate the region as *active*. When the region is active, **readline** highlights the text in the region using the value of the **active−region−start−color** variable, which defaults to the string that enables the terminal's standout mode. The active region shows the text inserted by bracketed-paste and any matching text found by incremental and non-incremental history searches.

enable−bracketed−paste (On)
When set to **On**, **readline** configures the terminal to insert each paste into the editing buffer as a single string of characters, instead of treating each character as if it had been read from the keyboard. This is called *bracketed−paste mode*; it prevents **readline** from executing any editing

commands bound to key sequences appearing in the pasted text.

enable−keypad (Off)
When set to **On**, **readline** tries to enable the application keypad when it is called. Some systems need this to enable the arrow keys.

enable−meta−key (On)
When set to **On**, **readline** tries to enable any meta modifier key the terminal claims to support. On many terminals, the Meta key is used to send eight-bit characters; this variable checks for the terminal capability that indicates the terminal can enable and disable a mode that sets the eighth bit of a character (0200) if the Meta key is held down when the character is typed (a meta character).

expand−tilde (Off)
If set to **On**, **readline** performs tilde expansion when it attempts word completion.

force−meta−prefix (Off)
If set to **On**, **readline** modifies its behavior when binding key sequences containing \M- or Meta- (see **Key Bindings** above) by converting a key sequence of the form \M−*C* or Meta−*C* to the two-character sequence **ESC** *C* (adding the meta prefix). If **force−meta−prefix** is set to **Off** (the default), **readline** uses the value of the **convert−meta** variable to determine whether to perform this conversion: if **convert−meta** is **On**, **readline** performs the conversion described above; if it is **Off**, **readline** converts *C* to a meta character by setting the eighth bit (0200).

history−preserve−point (Off)
If set to **On**, the history code attempts to place point at the same location on each history line retrieved with **previous-history** or **next-history**.

history−size (unset)
Set the maximum number of history entries saved in the history list. If set to zero, any existing history entries are deleted and no new entries are saved. If set to a value less than zero, the number of history entries is not limited. By default, **bash** sets the maximum number of history entries to the value of the **HISTSIZE** shell variable. Setting *history−size* to a non-numeric value will set the maximum number of history entries to 500.

horizontal−scroll−mode (Off)
Setting this variable to **On** makes **readline** use a single line for display, scrolling the input horizontally on a single screen line when it becomes longer than the screen width rather than wrapping to a new line. This setting is automatically enabled for terminals of height 1.

input−meta (Off)
If set to **On**, **readline** enables eight-bit input (that is, it does not clear the eighth bit in the characters it reads), regardless of what the terminal claims it can support. The default is *Off*, but **readline** sets it to *On* if the locale contains characters whose encodings may include bytes with the eighth bit set. This variable is dependent on the **LC_CTYPE** locale category, and its value may change if the locale changes. The name **meta−flag** is a synonym for **input−meta**.

isearch−terminators (“**C−[C−j**”)
The string of characters that should terminate an incremental search without subsequently executing the character as a command. If this variable has not been given a value, the characters *ESC* and **C−j** terminate an incremental search.

keymap (emacs)
Set the current **readline** keymap. The set of valid keymap names is *emacs, emacs−standard, emacs−meta, emacs−ctlx, vi, vi−command*, and *vi−insert*. *vi* is equivalent to *vi−command*; *emacs* is equivalent to *emacs−standard*. The default value is *emacs*; the value of **editing−mode** also affects the default keymap.

keyseq−timeout (500)
Specifies the duration **readline** will wait for a character when reading an ambiguous key sequence (one that can form a complete key sequence using the input read so far, or can take additional input to complete a longer key sequence). If **readline** does not receive any input within the timeout, it uses the shorter but complete key sequence. The value is specified in milliseconds, so a value of 1000 means that **readline** will wait one second for additional input. If this variable is set to a value less than or equal to zero, or to a non-numeric value, **readline** waits until another key is pressed to decide which key sequence to complete.

mark−directories (On)
If set to **On**, completed directory names have a slash appended.

mark−modified−lines (Off)
If set to **On**, **readline** displays history lines that have been modified with a preceding asterisk (*****).

mark−symlinked−directories (Off)
If set to **On**, completed names which are symbolic links to directories have a slash appended, subject to the value of **mark−directories**.

match−hidden−files (On)
This variable, when set to **On**, forces **readline** to match files whose names begin with a "." (hidden files) when performing filename completion. If set to **Off**, the user must include the leading "." in the filename to be completed.

menu−complete−display−prefix (Off)
If set to **On**, menu completion displays the common prefix of the list of possible completions (which may be empty) before cycling through the list.

output−meta (Off)
If set to **On**, **readline** displays characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is *Off*, but **readline** sets it to *On* if the locale contains characters whose encodings may include bytes with the eighth bit set. This variable is dependent on the **LC_CTYPE** locale category, and its value may change if the locale changes.

page−completions (On)
If set to **On**, **readline** uses an internal pager resembling *more*(1) to display a screenful of possible completions at a time.

prefer−visible−bell
See **bell−style**.

print−completions−horizontally (Off)
If set to **On**, **readline** displays completions with matches sorted horizontally in alphabetical order, rather than down the screen.

revert−all−at−newline (Off)
If set to **On**, **readline** will undo all changes to history lines before returning when executing **accept−line**. By default, history lines may be modified and retain individual undo lists across calls to **readline**.

search−ignore−case (Off)
If set to **On**, **readline** performs incremental and non-incremental history list searches in a case−insensitive fashion.

show−all−if−ambiguous (Off)
This alters the default behavior of the completion functions. If set to **On**, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell.

show−all−if−unmodified (Off)
This alters the default behavior of the completion functions in a fashion similar to **show−all−if−ambiguous**. If set to **On**, words which have more than one possible completion without any possible partial completion (the possible completions don't share a common prefix) cause the matches to be listed immediately instead of ringing the bell.

show−mode−in−prompt (Off)
If set to **On**, add a string to the beginning of the prompt indicating the editing mode: emacs, vi command, or vi insertion. The mode strings are user-settable (e.g., *emacs−mode−string*).

skip−completed−text (Off)
If set to **On**, this alters the default completion behavior when inserting a single match into the line. It's only active when performing completion in the middle of a word. If enabled, **readline** does not insert characters from the completion that match characters after point in the word being completed, so portions of the word following the cursor are not duplicated.

vi−cmd−mode−string ((cmd))
If the *show−mode−in−prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in command mode. The value

is expanded like a key binding, so the standard set of meta- and control- prefixes and backslash escape sequences is available. The \1 and \2 escapes begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string.

vi−ins−mode−string ((ins))

If the *show−mode−in−prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in insertion mode. The value is expanded like a key binding, so the standard set of meta- and control- prefixes and backslash escape sequences is available. The \1 and \2 escapes begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string.

visible−stats (Off)

If set to **On**, a character denoting a file's type as reported by *stat*(2) is appended to the filename when listing possible completions.

Readline Conditional Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives available.

$if The **$if** construct allows bindings to be made based on the editing mode, the terminal being used, or the application using **readline**. The text of the test, after any comparison operator, extends to the end of the line; unless otherwise noted, no characters are required to isolate it.

mode The **mode=** form of the **$if** directive is used to test whether **readline** is in emacs or vi mode. This may be used in conjunction with the **set keymap** command, for instance, to set bindings in the *emacs−standard* and *emacs−ctlx* keymaps only if **readline** is starting out in emacs mode.

term The **term=** form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the **=** is tested against both the full name of the terminal and the portion of the terminal name before the first **−**. This allows *xterm* to match both *xterm* and *xterm−256color*, for instance.

version

The **version** test may be used to perform comparisons against specific **readline** versions. The **version** expands to the current **readline** version. The set of comparison operators includes **=**, (and **==**), **!=**, **<=**, **>=**, **<**, and **>**. The version number supplied on the right side of the operator consists of a major version number, an optional decimal point, and an optional minor version (e.g., **7.1**). If the minor version is omitted, it defaults to **0**. The operator may be separated from the string **version** and from the version number argument by whitespace.

application

The *application* construct is used to include application-specific settings. Each program using the **readline** library sets the *application name*, and an initialization file can test for a particular value. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in **bash**:

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

variable

The *variable* construct provides simple equality tests for **readline** variables and values. The permitted comparison operators are *=*, *==*, and *!=*. The variable name must be separated from the comparison operator by whitespace; the operator may be separated from the value on the right hand side by whitespace. String and boolean variables may be tested. Boolean variables must be tested against the values *on* and *off*.

$else Commands in this branch of the **$if** directive are executed if the test fails.

$endif This command, as seen in the previous example, terminates an **$if** command.

$include
This directive takes a single filename as an argument and reads commands and key bindings from that file. For example, the following directive would read */etc/inputrc*:

$include */etc/inputrc*

Searching

Readline provides commands for searching through the command history (see **HISTORY** below) for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, **readline** displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. When using emacs editing mode, type **C−r** to search backward in the history for a particular string. Typing **C−s** searches forward through the history. The characters present in the value of the **isearch-terminators** variable are used to terminate an incremental search. If that variable has not been assigned a value, *ESC* and **C−j** terminate an incremental search. **C−g** aborts an incremental search and restores the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type **C−r** or **C−s** as appropriate. This searches backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a **readline** command terminates the search and executes that command. For instance, a newline terminates the search and accepts the line, thereby executing the command from the history list. A movement command will terminate the search, make the last line found the current line, and begin editing.

Readline remembers the last incremental search string. If two **C−r**s are typed without any intervening characters defining a new search string, **readline** uses any remembered search string.

Non-incremental searches read the entire search string before starting to search for matching history entries. The search string may be typed by the user or be part of the contents of the current line.

Readline Command Names

The following is a list of the names of the commands and the default key sequences to which they are bound. Command names without an accompanying key sequence are unbound by default.

In the following descriptions, *point* refers to the current cursor position, and *mark* refers to a cursor position saved by the **set−mark** command. The text between the point and mark is referred to as the *region*. **Readline** has the concept of an *active region*: when the region is active, **readline** redisplay highlights the region using the value of the **active−region−start−color** variable. The **enable−active−region readline** variable turns this on and off. Several commands set the region to active; those are noted below.

Commands for Moving

beginning−of−line (C−a)
Move to the start of the current line. This may also be bound to the Home key on some keyboards.

end−of−line (C−e)
Move to the end of the line. This may also be bound to the End key on some keyboards.

forward−char (C−f)
Move forward a character. This may also be bound to the right arrow key on some keyboards.

backward−char (C−b)
Move back a character. This may also be bound to the left arrow key on some keyboards.

forward−word (M−f)
Move forward to the end of the next word. Words are composed of alphanumeric characters (letters and digits).

backward−word (M−b)
Move back to the start of the current or previous word. Words are composed of alphanumeric characters (letters and digits).

shell−forward−word (M−C−f)
Move forward to the end of the next word. Words are delimited by non-quoted shell metacharacters.

shell−backward−word (M−C−b)
Move back to the start of the current or previous word. Words are delimited by non-quoted shell metacharacters.

previous−screen−line
Attempt to move point to the same physical screen column on the previous physical screen line. This will not have the desired effect if the current **readline** line does not take up more than one physical line or if point is not greater than the length of the prompt plus the screen width.

next−screen−line
Attempt to move point to the same physical screen column on the next physical screen line. This will not have the desired effect if the current **readline** line does not take up more than one physical line or if the length of the current **readline** line is not greater than the length of the prompt plus the screen width.

clear−display (M−C−l)
Clear the screen and, if possible, the terminal's scrollback buffer, then redraw the current line, leaving the current line at the top of the screen.

clear−screen (C−l)
Clear the screen, then redraw the current line, leaving the current line at the top of the screen. With a numeric argument, refresh the current line without clearing the screen.

redraw−current−line
Refresh the current line.

Commands for Manipulating the History

accept−line (Newline, Return)
Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the state of the **HISTCONTROL** and **HISTIGNORE** variables. If the line is a modified history line, restore the history line to its original state.

previous−history (C−p)
Fetch the previous command from the history list, moving back in the list. This may also be bound to the up arrow key on some keyboards.

next−history (C−n)
Fetch the next command from the history list, moving forward in the list. This may also be bound to the down arrow key on some keyboards.

beginning−of−history (M−<)
Move to the first line in the history.

end−of−history (M−>)
Move to the end of the input history, i.e., the line currently being entered.

operate−and−get−next (C−o)
Accept the current line for execution as if a newline had been entered, and fetch the next line relative to the current line from the history for editing. A numeric argument, if supplied, specifies the history entry to use instead of the current line.

fetch−history
With a numeric argument, fetch that entry from the history list and make it the current line. Without an argument, move back to the first entry in the history list.

reverse−search−history (C−r)
Search backward starting at the current line and moving "up" through the history as necessary. This is an incremental search. This command sets the region to the matched text and activates the region.

forward−search−history (C−s)
Search forward starting at the current line and moving "down" through the history as necessary. This is an incremental search. This command sets the region to the matched text and activates the region.

non−incremental−reverse−search−history (M−p)
Search backward through the history starting at the current line using a non-incremental search for a string supplied by the user. The search string may match anywhere in a history line.

non−incremental−forward−search−history (M−n)
Search forward through the history using a non-incremental search for a string supplied by the user. The search string may match anywhere in a history line.

history−search−backward
Search backward through the history for the string of characters between the start of the current line and the point. The search string must match at the beginning of a history line. This is a non-incremental search. This may be bound to the Page Up key on some keyboards.

history−search−forward
Search forward through the history for the string of characters between the start of the current line and the point. The search string must match at the beginning of a history line. This is a non-incremental search. This may be bound to the Page Down key on some keyboards.

history−substring−search−backward
Search backward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere in a history line. This is a non-incremental search.

history−substring−search−forward
Search forward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere in a history line. This is a non-incremental search.

yank−nth−arg (M−C−y)
Insert the first argument to the previous command (usually the second word on the previous line) at point. With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command. Once the argument *n* is computed, this uses the history expansion facilities to extract the *n*th word, as if the "!*n*" history expansion had been specified.

yank−last−arg (M−., M−_)
Insert the last argument to the previous command (the last word of the previous history entry). With a numeric argument, behave exactly like **yank−nth−arg**. Successive calls to **yank−last−arg** move back through the history list, inserting the last word (or the word specified by the argument to the first call) of each line in turn. Any numeric argument supplied to these successive calls determines the direction to move through the history. A negative argument switches the direction through the history (back or forward). This uses the history expansion facilities to extract the last word, as if the "!$" history expansion had been specified.

shell−expand−line (M−C−e)
Expand the line by performing shell word expansions. This performs alias and history expansion, **$**'*string*' and **$**"*string*" quoting, tilde expansion, parameter and variable expansion, arithmetic expansion, command and process substitution, word splitting, and quote removal. An explicit argument suppresses command and process substitution. See **HISTORY EXPANSION** below for a description of history expansion.

history−expand−line (M−^)
Perform history expansion on the current line. See **HISTORY EXPANSION** below for a description of history expansion.

magic−space
Perform history expansion on the current line and insert a space. See **HISTORY EXPANSION** below for a description of history expansion.

alias−expand−line
Perform alias expansion on the current line. See **ALIASES** above for a description of alias expansion.

history−and−alias−expand−line
Perform history and alias expansion on the current line.

insert−last−argument (M−., M−_)
A synonym for **yank−last−arg**.

edit−and−execute−command (C−x C−e)
Invoke an editor on the current command line, and execute the result as shell commands. **Bash** attempts to invoke **$VISUAL**, **$EDITOR**, and *emacs* as the editor, in that order.

Commands for Changing Text

end−of−file **(usually C−d)**
The character indicating end-of-file as set, for example, by *stty*(1). If this character is read when there are no characters on the line, and point is at the beginning of the line, **readline** interprets it as the end of input and returns **EOF**.

delete−char (C−d)
Delete the character at point. If this function is bound to the same character as the tty **EOF** character, as **C−d** commonly is, see above for the effects. This may also be bound to the Delete key on some keyboards.

backward−delete−char (Rubout)
Delete the character behind the cursor. When given a numeric argument, save the deleted text on the kill ring.

forward−backward−delete−char
Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted.

quoted−insert (C−q, C−v)
Add the next character typed to the line verbatim. This is how to insert characters like **C−q**, for example.

tab−insert (C−v TAB)
Insert a tab character.

self−insert (a, b, A, 1, !, . . .**)**
Insert the character typed.

bracketed−paste−begin
This function is intended to be bound to the "bracketed paste" escape sequence sent by some terminals, and such a binding is assigned by default. It allows **readline** to insert the pasted text as a single unit without treating each character as if it had been read from the keyboard. The pasted characters are inserted as if each one was bound to **self−insert** instead of executing any editing commands.
Bracketed paste sets the region to the inserted text and activates the region.

transpose−chars (C−t)
Drag the character before point forward over the character at point, moving point forward as well. If point is at the end of the line, then this transposes the two characters before point. Negative arguments have no effect.

transpose−words (M−t)
Drag the word before point past the word after point, moving point past that word as well. If point is at the end of the line, this transposes the last two words on the line.

shell−transpose−words (M-C-t)
Drag the word before point past the word after point, moving point past that word as well. If the insertion point is at the end of the line, this transposes the last two words on the line. Word boundaries are the same as **shell−forward−word** and **shell−backward−word**.

upcase−word (M−u)
Uppercase the current (or following) word. With a negative argument, uppercase the previous word, but do not move point.

downcase−word (M−l)
Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move point.

capitalize−word (M−c)
Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move point.

overwrite−mode
Toggle overwrite mode. With an explicit positive numeric argument, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only **emacs** mode; **vi** mode does overwrite differently. Each call to *readline()* starts in insert mode.
In overwrite mode, characters bound to **self−insert** replace the text at point rather than pushing the text to the right. Characters bound to **backward−delete−char** replace the character before point with a space. By default, this command is unbound, but may be bound to the Insert key on some keyboards.

Killing and Yanking

kill−line (C−k)
Kill the text from point to the end of the current line. With a negative numeric argument, kill backward from the cursor to the beginning of the line.

backward−kill−line (C−x Rubout)
Kill backward to the beginning of the current line. With a negative numeric argument, kill forward from the cursor to the end of the line.

unix−line−discard (C−u)
Kill backward from point to the beginning of the line, saving the killed text on the kill-ring.

kill−whole−line
Kill all characters on the current line, no matter where point is.

kill−word (M−d)
Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by **forward−word**.

backward−kill−word (M−Rubout)
Kill the word behind point. Word boundaries are the same as those used by **backward−word**.

shell−kill−word (M−C−d)
Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by **shell−forward−word**.

shell−backward−kill−word
Kill the word behind point. Word boundaries are the same as those used by **shell−backward−word**.

unix−word−rubout (C−w)
Kill the word behind point, using white space as a word boundary, saving the killed text on the kill-ring.

unix−filename−rubout
Kill the word behind point, using white space and the slash character as the word boundaries, saving the killed text on the kill-ring.

delete−horizontal−space (M−\)
Delete all spaces and tabs around point.

kill−region
Kill the text in the current region.

copy−region−as−kill
Copy the text in the region to the kill buffer, so it can be yanked immediately.

copy−backward−word
Copy the word before point to the kill buffer. The word boundaries are the same as **backward−word**.

copy−forward−word
Copy the word following point to the kill buffer. The word boundaries are the same as **forward−word**.

yank (C−y)
Yank the top of the kill ring into the buffer at point.

yank−pop (M−y)
Rotate the kill ring, and yank the new top. Only works following **yank** or **yank−pop**.

Numeric Arguments

digit−argument (M−0, M−1, ..., M−−)
Add this digit to the argument already accumulating, or start a new argument. M−− starts a negative argument.

universal−argument
This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing **universal−argument** again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit nor minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on.

Completing

complete (TAB)
Attempt to perform completion on the text before point. **Bash** attempts completion by first checking for any programmable completions for the command word (see **Programmable Completion** below), otherwise treating the text as a variable (if the text begins with **$**), username (if the text begins with **~**), hostname (if the text begins with **@**), or command (including aliases, functions, and builtins) in turn. If none of these produces a match, it falls back to filename completion.

possible−completions (M−?)
List the possible completions of the text before point. When displaying completions, **readline** sets the number of columns used for display to the value of **completion-display-width**, the value of the shell variable **COLUMNS**, or the screen width, in that order.

insert−completions (M−*)
Insert all completions of the text before point that would have been generated by **possible−completions**, separated by a space.

menu−complete
Similar to **complete**, but replaces the word to be completed with a single match from the list of possible completions. Repeatedly executing **menu−complete** steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, **menu−complete** rings the bell (subject to the setting of **bell−style**) and restores the original text. An argument of *n* moves *n* positions forward in the list of matches; a negative argument moves backward through the list. This command is intended to be bound to **TAB**, but is unbound by default.

menu−complete−backward
Identical to **menu−complete**, but moves backward through the list of possible completions, as if **menu−complete** had been given a negative argument. This command is unbound by default.

export−completions
Perform completion on the word before point as described above and write the list of possible completions to **readline**'s output stream using the following format, writing information on separate lines:

- the number of matches *N*;
- the word being completed;
- *S*:*E*, where *S* and *E* are the start and end offsets of the word in the **readline** line buffer; then
- each match, one per line

If there are no matches, the first line will be "0", and this command does not print any output after the *S*:*E*. If there is only a single match, this prints a single line containing it. If there is more than one match, this prints the common prefix of the matches, which may be empty, on the first line after the *S*:*E*, then the matches on subsequent lines. In this case, *N* will include the first line with the common prefix.

The user or application should be able to accommodate the possibility of a blank line. The intent is that the user or application reads *N* lines after the line containing *S*:*E* to obtain the match list. This command is unbound by default.

delete−char−or−list
Deletes the character under the cursor if not at the beginning or end of the line (like **delete−char**). At the end of the line, it behaves identically to **possible−completions**. This command is unbound by default.

complete−filename (M−/)
Attempt filename completion on the text before point.

possible−filename−completions (C−x /)
List the possible completions of the text before point, treating it as a filename.

complete−username (M−~)
Attempt completion on the text before point, treating it as a username.

possible−username−completions (C−x ~)
List the possible completions of the text before point, treating it as a username.

complete−variable (M−$)
Attempt completion on the text before point, treating it as a shell variable.

possible−variable−completions (C−x $)
List the possible completions of the text before point, treating it as a shell variable.

complete−hostname (M−@)
Attempt completion on the text before point, treating it as a hostname.

possible−hostname−completions (C−x @)
List the possible completions of the text before point, treating it as a hostname.

complete−command (M−!)
Attempt completion on the text before point, treating it as a command name. Command completion attempts to match the text against aliases, reserved words, shell functions, shell builtins, and finally executable filenames, in that order.

possible−command−completions (C−x !)
List the possible completions of the text before point, treating it as a command name.

dynamic−complete−history (M−TAB)
Attempt completion on the text before point, comparing the text against history list entries for possible completion matches.

dabbrev−expand
Attempt menu completion on the text before point, comparing the text against lines from the history list for possible completion matches.

complete−into−braces (M−{)
Perform filename completion and insert the list of possible completions enclosed within braces so the list is available to the shell (see **Brace Expansion** above).

Keyboard Macros

start−kbd−macro (C−x ()
Begin saving the characters typed into the current keyboard macro.

end−kbd−macro (C−x))
Stop saving the characters typed into the current keyboard macro and store the definition.

call−last−kbd−macro (C−x e)
Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

print−last−kbd−macro ()
Print the last keyboard macro defined in a format suitable for the *inputrc* file.

Miscellaneous

re−read−init−file (C−x C−r)
Read in the contents of the *inputrc* file, and incorporate any bindings or variable assignments found there.

abort (C−g)
Abort the current editing command and ring the terminal's bell (subject to the setting of **bell−style**).

do−lowercase−version (M−A, M−B, M−*x*, . . .)
If the metafied character *x* is uppercase, run the command that is bound to the corresponding metafied lowercase character. The behavior is undefined if *x* is already lowercase.

prefix−meta (ESC)
Metafy the next character typed. **ESC f** is equivalent to **Meta−f**.

undo (C−_, C−x C−u)
Incremental undo, separately remembered for each line.

revert−line (M−r)
Undo all changes made to this line. This is like executing the **undo** command enough times to return the line to its initial state.

tilde−expand (M−&)
Perform tilde expansion on the current word.

set−mark (C−@, M−<space>)
Set the mark to the point. If a numeric argument is supplied, set the mark to that position.

exchange−point−and−mark (C−x C−x)
Swap the point with the mark. Set the current cursor position to the saved position, then set the mark to the old cursor position.

character−search (C−])
Read a character and move point to the next occurrence of that character. A negative argument searches for previous occurrences.

character−search−backward (M−C−])
Read a character and move point to the previous occurrence of that character. A negative argument searches for subsequent occurrences.

skip−csi−sequence
Read enough characters to consume a multi-key sequence such as those defined for keys like Home and End. CSI sequences begin with a Control Sequence Indicator (CSI), usually *ESC [*. If this sequence is bound to "\e[", keys producing CSI sequences have no effect unless explicitly bound to a **readline** command, instead of inserting stray characters into the editing buffer. This is unbound by default, but usually bound to *ESC [*.

insert−comment (M−#)
Without a numeric argument, insert the value of the **readline comment−begin** variable at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of **comment−begin**, insert the value; otherwise delete the characters in **comment-begin** from the beginning of the line. In either case, the line is accepted as if a newline had been typed. The default value of **comment−begin** causes this command to make the current line a shell comment. If a numeric argument causes the comment character to be removed, the line will be executed by the shell.

spell−correct−word (C−x s)
Perform spelling correction on the current word, treating it as a directory or filename, in the same way as the **cdspell** shell option. Word boundaries are the same as those used by **shell−forward−word**.

glob−complete−word (M−g)
Treat the word before point as a pattern for pathname expansion, with an asterisk implicitly appended, then use the pattern to generate a list of matching file names for possible completions.

glob−expand−word (C−x *)
Treat the word before point as a pattern for pathname expansion, and insert the list of matching file names, replacing the word. If a numeric argument is supplied, append a * before pathname expansion.

glob−list−expansions (C−x g)
Display the list of expansions that would have been generated by **glob−expand−word** and redisplay the line. If a numeric argument is supplied, append a * before pathname expansion.

dump−functions
Print all of the functions and their key bindings to the **readline** output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

dump−variables
Print all of the settable **readline** variables and their values to the **readline** output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

dump−macros
Print all of the **readline** key sequences bound to macros and the strings they output to the **readline** output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.

execute−named−command (M-x)
Read a bindable **readline** command name from the input and execute the function to which it's bound, as if the key sequence to which it was bound appeared in the input. If this function is supplied with a numeric argument, it passes that argument to the function it executes.

display−shell−version (C−x C−v)
Display version information about the current instance of **bash**.

Programmable Completion

When a user attempts word completion for a command or an argument to a command for which a completion specification (a *compspec*) has been defined using the **complete** builtin (see **SHELL BUILTIN COMMANDS** below), **readline** invokes the programmable completion facilities.

First, **bash** identifies the command name. If a compspec has been defined for that command, the compspec is used to generate the list of possible completions for the word. If the command word is the empty string (completion attempted at the beginning of an empty line), **bash** uses any compspec defined with the **−E** option to **complete**. The **−I** option to **complete** indicates that the command word is the first non-assignment word on the line, or after a command delimiter such as **;** or **|**. This usually indicates command name completion.

If the command word is a full pathname, **bash** searches for a compspec for the full pathname first. If there is no compspec for the full pathname, **bash** attempts to find a compspec for the portion following the final slash. If those searches do not result in a compspec, or if there is no compspec for the command word, **bash** uses any compspec defined with the **−D** option to **complete** as the default. If there is no default compspec, **bash** performs alias expansion on the command word as a final resort, and attempts to find a compspec for the command word resulting from any successful expansion.

If a compspec is not found, **bash** performs its default completion as described above under **Completing**. Otherwise, once a compspec has been found, **bash** uses it to generate the list of matching words.

First, **bash** performs the *actions* specified by the compspec. This only returns matches which are prefixes of the word being completed. When the **−f** or **−d** option is used for filename or directory name completion, **bash** uses the shell variable **FIGNORE** to filter the matches.

Next, programmable completion generates matches specified by a pathname expansion pattern supplied as an argument to the **−G** option. The words generated by the pattern need not match the word being completed. **Bash** uses the **FIGNORE** variable to filter the matches, but does not use the **GLOBIGNORE** shell variable.

Next, completion considers the string specified as the argument to the **−W** option. The string is first split using the characters in the **IFS** special variable as delimiters. This honors shell quoting within the string, in order to provide a mechanism for the words to contain shell metacharacters or characters in the value of **IFS**. Each word is then expanded using brace expansion, tilde expansion, parameter and variable expansion, command substitution, and arithmetic expansion, as described above under **EXPANSION**. The results are split using the rules described above under **Word Splitting**. The results of the expansion are prefix-matched against the word being completed, and the matching words become possible completions.

After these matches have been generated, **bash** executes any shell function or command specified with the **−F** and **−C** options. When the command or function is invoked, **bash** assigns values to the **COMP_LINE**, **COMP_POINT**, **COMP_KEY**, and **COMP_TYPE** variables as described above under **Shell Variables**. If a shell function is being invoked, **bash** also sets the **COMP_WORDS** and **COMP_CWORD** variables. When the function or command is invoked, the first argument (**$1**) is the name of the command whose arguments are being completed, the second argument (**$2**) is the word being completed, and the third argument (**$3**) is the word preceding the word being completed on the current command line. There is no filtering of the generated completions against the word being completed; the function or command has complete freedom in generating the matches and they do not need to match a prefix of the word.

Any function specified with **−F** is invoked first. The function may use any of the shell facilities, including the **compgen** and **compopt** builtins described below, to generate the matches. It must put the possible completions in the **COMPREPLY** array variable, one per array element.

Next, any command specified with the **−C** option is invoked in an environment equivalent to command substitution. It should print a list of completions, one per line, to the standard output. Backslash will escape a newline, if necessary. These are added to the set of possible completions.

After generating all of the possible completions, **bash** applies any filter specified with the **−X** option to the completions in the list. The filter is a pattern as used for pathname expansion; a **&** in the pattern is replaced with the text of the word being completed. A literal **&** may be escaped with a backslash; the backslash is removed before attempting a match. Any completion that matches the pattern is removed from the list. A leading **!** negates the pattern; in this case **bash** removes any completion that does not match the pattern. If the **nocasematch** shell option is enabled, **bash** performs the match without regard to the case of alphabetic characters.

Finally, programmable completion adds any prefix and suffix specified with the **−P** and **−S** options, respectively, to each completion, and returns the result to **readline** as the list of possible completions.

If the previously-applied actions do not generate any matches, and the **−o dirnames** option was supplied to **complete** when the compspec was defined, **bash** attempts directory name completion.

If the **−o plusdirs** option was supplied to **complete** when the compspec was defined, **bash** attempts directory name completion and adds any matches to the set of possible completions.

By default, if a compspec is found, whatever it generates is returned to the completion code as the full set of possible completions. The default **bash** completions and the **readline** default of filename completion are disabled. If the **−o bashdefault** option was supplied to **complete** when the compspec was defined, and the compspec generates no matches, **bash** attempts its default completions. If the compspec and, if attempted, the default **bash** completions generate no matches, and the **−o default** option was supplied to **complete** when the compspec was defined, programmable completion performs **readline**'s default completion.

The options supplied to **complete** and **compopt** can control how **readline** treats the completions. For instance, the *−o fullquote* option tells **readline** to quote the matches as if they were filenames. See the description of **complete** below for details.

When a compspec indicates that it wants directory name completion, the programmable completion functions force **readline** to append a slash to completed names which are symbolic links to directories, subject to the value of the **mark−directories readline** variable, regardless of the setting of the **mark-symlinked−directories readline** variable.

There is some support for dynamically modifying completions. This is most useful when used in combination with a default completion specified with **complete −D**. It's possible for shell functions executed as completion functions to indicate that completion should be retried by returning an exit status of 124. If a shell function returns 124, and changes the compspec associated with the command on which completion is being attempted (supplied as the first argument when the function is executed), programmable completion restarts from the beginning, with an attempt to find a new compspec for that command. This can be used to build a set of completions dynamically as completion is attempted, rather than loading them all at once.

For instance, assuming that there is a library of compspecs, each kept in a file corresponding to the name of the command, the following default completion function would load completions dynamically:

```
_completion_loader()
{
  . "/etc/bash_completion.d/$1.sh" >/dev/null 2>&1 && return 124
}
complete -D -F _completion_loader -o bashdefault -o default
```

HISTORY

When the **−o history** option to the **set** builtin is enabled, the shell provides access to the *command history*, the list of commands previously typed. The value of the **HISTSIZE** variable is used as the number of commands to save in a history list: the shell saves the text of the last **HISTSIZE** commands (default 500). The shell stores each command in the history list prior to parameter and variable expansion (see **EXPANSION** above) but after history expansion is performed, subject to the values of the shell variables **HISTIGNORE** and **HISTCONTROL**.

On startup, **bash** initializes the history list by reading history entries from the file named by the **HISTFILE** variable (default *~/.bash_history*). That file is referred to as the *history file*. The history file is truncated, if necessary, to contain no more than the number of history entries specified by the value of the **HISTFILESIZE** variable. If **HISTFILESIZE** is unset, or set to null, a non-numeric value, or a numeric value less than zero, the history file is not truncated.

When the history file is read, lines beginning with the history comment character followed immediately by a digit are interpreted as timestamps for the following history line. These timestamps are optionally displayed depending on the value of the **HISTTIMEFORMAT** variable. When present, history timestamps delimit history entries, making multi-line entries possible.

When a shell with history enabled exits, **bash** copies the last **$HISTSIZE** entries from the history list to **$HISTFILE**. If the **histappend** shell option is enabled (see the description of **shopt** under **SHELL BUILTIN COMMANDS** below), **bash** appends the entries to the history file, otherwise it overwrites the history file. If **HISTFILE** is unset or null, or if the history file is unwritable, the history is not saved. After saving the history, **bash** truncates the history file to contain no more than **HISTFILESIZE** lines as described above.

If the **HISTTIMEFORMAT** variable is set, the shell writes the timestamp information associated with each history entry to the history file, marked with the history comment character, so timestamps are preserved across shell sessions. This uses the history comment character to distinguish timestamps from other history lines. As above, when using **HISTTIMEFORMAT**, the timestamps delimit multi-line history entries.

The **fc** builtin command (see **SHELL BUILTIN COMMANDS** below) will list or edit and re-execute a portion of the history list. The **history** builtin can display or modify the history list and manipulate the history file. When using command-line editing, search commands are available in each editing mode that provide access to the history list.

The shell allows control over which commands are saved on the history list. The **HISTCONTROL** and **HISTIGNORE** variables are used to save only a subset of the commands entered. If the **cmdhist** shell option is enabled, the shell attempts to save each line of a multi-line command in the same history entry, adding semicolons where necessary to preserve syntactic correctness. The **lithist** shell option modifies **cmdhist** by saving the command with embedded newlines instead of semicolons. See the description of the **shopt** builtin below under **SHELL BUILTIN COMMANDS** for information on setting and unsetting shell options.

HISTORY EXPANSION

The shell supports a history expansion feature that is similar to the history expansion in **csh**. This section describes what syntax features are available.

History expansion is enabled by default for interactive shells, and can be disabled using the **+H** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below). Non-interactive shells do not perform history expansion by default, but it can be enabled with “set -H”.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words, and is performed on each line individually. The shell attempts to inform the history expansion functions about quoting still in effect from previous lines.

It takes place in two parts. The first is to determine which history list entry to use during substitution. The second is to select portions of that entry to include into the current one.

The entry selected from the history is the *event*, and the portions of that entry that are acted upon are *words*. Various *modifiers* are available to manipulate the selected words. The entry is split into words in the same fashion as when reading input, so that several *metacharacter*-separated words surrounded by quotes are considered one word. The *event designator* selects the event, the optional *word designator* selects words from the event, and various optional *modifiers* are available to manipulate the selected words.

History expansions are introduced by the appearance of the history expansion character, which is **!** by default. History expansions may appear anywhere in the input, but do not nest.

Only backslash (****) and single quotes can quote the history expansion character, but the history expansion character is also treated as quoted if it immediately precedes the closing double quote in a double-quoted string.

Several characters inhibit history expansion if found immediately following the history expansion character, even if it is unquoted: space, tab, newline, carriage return, **=**, and the other shell metacharacters defined above.

There is a special abbreviation for substitution, active when the *quick substitution* character (described above under **histchars**) is the first character on the line. It selects the previous history list entry, using an event designator equivalent to **!!**, and substitutes one string for another in that entry. It is described below under **Event Designators**. This is the only history expansion that does not begin with the history expansion character.

Several shell options settable with the **shopt** builtin will modify history expansion behavior (see the description of the **shopt** builtin below).and If the **histverify** shell option is enabled, and **readline** is being used, history substitutions are not immediately passed to the shell parser. Instead, the expanded line is reloaded into the **readline** editing buffer for further modification. If **readline** is being used, and the **histreedit** shell option is enabled, a failed history substitution is reloaded into the **readline** editing buffer for correction.

The **−p** option to the **history** builtin command shows what a history expansion will do before using it. The **−s** option to the **history** builtin will add commands to the end of the history list without actually executing them, so that they are available for subsequent recall.

The shell allows control of the various characters used by the history expansion mechanism (see the description of **histchars** above under **Shell Variables**). The shell uses the history comment character to mark history timestamps when writing the history file.

Event Designators

An event designator is a reference to an entry in the history list. The event designator consists of the portion of the word beginning with the history expansion character and ending with the word designator if present, or the end of the word. Unless the reference is absolute, events are relative to the current position in the history list.

! Start a history substitution, except when followed by a **blank**, newline, carriage return, =, or, when the **extglob** shell option is enabled using the **shopt** builtin, (.

!*n* Refer to history list entry *n*.

!−*n* Refer to the current entry minus *n*.

!! Refer to the previous entry. This is a synonym for "!−1".

!*string* Refer to the most recent command preceding the current position in the history list starting with *string*.

!?*string***[?]**
Refer to the most recent command preceding the current position in the history list containing *string*. The trailing **?** may be omitted if *string* is followed immediately by a newline. If *string* is

missing, this uses the string from the most recent search; it is an error if there is no previous search string.

^*string1*^*string2*^
: Quick substitution. Repeat the previous command, replacing *string1* with *string2*. Equivalent to "!!:s^*string1*^*string2*^" (see **Modifiers** below).

!# The entire command line typed so far.

Word Designators

Word designators are used to select desired words from the event. They are optional; if the word designator isn't supplied, the history expansion uses the entire event. A **:** separates the event specification from the word designator. It may be omitted if the word designator begins with a **^**, **$**, *****, **–**, or **%**. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line separated by single spaces.

0 (zero)
: The zeroth word. For the shell, this is the command word.

n The *n*th word.

^ The first argument: word 1.

$ The last word. This is usually the last argument, but will expand to the zeroth word if there is only one word in the line.

% The first word matched by the most recent "?*string*?" search, if the search string begins with a character that is part of a word. By default, searches begin at the end of each line and proceed to the beginning, so the first word matched is the one closest to the end of the line.

x–*y* A range of words; "–*y*" abbreviates "0–*y*".

***** All of the words but the zeroth. This is a synonym for "*1–$*". It is not an error to use ***** if there is just one word in the event; it expands to the empty string in that case.

x* Abbreviates *x–$*.

x– Abbreviates *x–$* like **x***, but omits the last word. If **x** is missing, it defaults to 0.

If a word designator is supplied without an event specification, the previous command is used as the event, equivalent to **!!**.

Modifiers

After the optional word designator, the expansion may include a sequence of one or more of the following modifiers, each preceded by a ":". These modify, or edit, the word or words selected from the history event.

h Remove a trailing pathname component, leaving only the head.

t Remove all leading pathname components, leaving the tail.

r Remove a trailing suffix of the form *.xxx*, leaving the basename.

e Remove all but the trailing suffix.

p Print the new command but do not execute it.

q Quote the substituted words, escaping further substitutions.

x Quote the substituted words as with **q**, but break into words at **blanks** and newlines. The **q** and **x** modifiers are mutually exclusive; expansion uses the last one supplied.

s/*old***/***new***/**
: Substitute *new* for the first occurrence of *old* in the event line. Any character may be used as the delimiter in place of /. The final delimiter is optional if it is the last character of the event line. A single backslash quotes the delimiter in *old* and *new*. If & appears in *new*, it is replaced with *old*. A single backslash quotes the &. If *old* is null, it is set to the last *old* substituted, or, if no previous history substitutions took place, the last *string* in a **!?***string***[?]** search. If *new* is null, each matching *old* is deleted.

& Repeat the previous substitution.

g Cause changes to be applied over the entire event line. This is used in conjunction with "**:s**" (e.g., "**:gs/***old***/***new***/**") or "**:&**". If used with "**:s**", any delimiter can be used in place of /, and the final delimiter is optional if it is the last character of the event line. An **a** may be used as a synonym for **g**.

G Apply the following “**s**” or “**&**” modifier once to each word in the event line.

SHELL BUILTIN COMMANDS

Unless otherwise noted, each builtin command documented in this section as accepting options preceded by **–** accepts **––** to signify the end of the options. The **:**, **true**, **false**, and **test/[** builtins do not accept options and do not treat **––** specially. The **exit**, **logout**, **return**, **break**, **continue**, **let**, and **shift** builtins accept and process arguments beginning with **–** without requiring **––**. Other builtins that accept arguments but are not specified as accepting options interpret arguments beginning with **–** as invalid options and require **––** to prevent this interpretation.

: [*arguments*]
No effect; the command does nothing beyond expanding *arguments* and performing any specified redirections. The return status is zero.

. [**–p** *path*] *filename* [*arguments*]
source [**–p** *path*] *filename* [*arguments*]
The **.** command (**source**) reads and execute commands from *filename* in the current shell environment and returns the exit status of the last command executed from *filename*.

If *filename* does not contain a slash, **.** searches for it. If the **–p** option is supplied, **.** treats *path* as a colon-separated list of directories in which to find *filename*; otherwise, **.** uses the entries in **PATH** to find the directory containing *filename*. *filename* does not need to be executable. When **bash** is not in posix mode, it searches the current directory if *filename* is not found in **PATH**, but does not search the current directory if **–p** is supplied. If the **sourcepath** option to the **shopt** builtin command is turned off, **.** does not search **PATH**.

If any *arguments* are supplied, they become the positional parameters when *filename* is executed. Otherwise the positional parameters are unchanged.

If the **–T** option is enabled, **.** inherits any trap on **DEBUG**; if it is not, any **DEBUG** trap string is saved and restored around the call to **.**, and **.** unsets the **DEBUG** trap while it executes. If **–T** is not set, and the sourced file changes the **DEBUG** trap, the new value persists after **.** completes. The return status is the status of the last command executed from *filename* (0 if no commands are executed), and non-zero if *filename* is not found or cannot be read.

alias [**–p**] [*name*[=*value*] ...]
With no arguments or with the **–p** option, **alias** prints the list of aliases in the form **alias** *name*=*value* on standard output. When arguments are supplied, define an alias for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution when the alias is expanded during command parsing. For each *name* in the argument list for which no *value* is supplied, print the name and value of the alias *name*. **alias** returns true unless a *name* is given (without a corresponding =*value*) for which no alias has been defined.

bg [*jobspec* ...]
Resume each suspended job *jobspec* in the background, as if it had been started with **&**. If *jobspec* is not present, the shell uses its notion of the *current job*. **bg** *jobspec* returns 0 unless run when job control is disabled or, when run with job control enabled, any specified *jobspec* was not found or was started without job control.

bind [**–m** *keymap*] [**–lsvSVX**]
bind [**–m** *keymap*] [**–q** *function*] [**–u** *function*] [**–r** *keyseq*]
bind [**–m** *keymap*] **–f** *filename*
bind [**–m** *keymap*] **–x** *keyseq*[:] *shell–command*
bind [**–m** *keymap*] *keyseq*:*function–name*
bind [**–m** *keymap*] **–p**|**–P** [*readline–command*]
bind [**–m** *keymap*] *keyseq*:*readline–command*
bind *readline-command-line*
Display current **readline** key and function bindings, bind a key sequence to a **readline** function or macro or to a shell command, or set a **readline** variable. Each non-option argument is a key binding or command as it would appear in a **readline** initialization file such as *.inputrc*, but each

binding or command must be passed as a separate argument; e.g., '"\C−x\C−r": re−read−init−file'. In the following descriptions, output available to be re-read is formatted as commands that would appear in a **readline** initialization file or that would be supplied as individual arguments to a **bind** command. Options, if supplied, have the following meanings:

−m *keymap*
: Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are *emacs, emacs−standard, emacs−meta, emacs−ctlx, vi, vi−move, vi−command*, and *vi−insert*. *vi* is equivalent to *vi−command* (*vi−move* is also a synonym); *emacs* is equivalent to *emacs−standard*.

−l List the names of all **readline** functions.

−p Display **readline** function names and bindings in such a way that they can be used as an argument to a subsequent **bind** command or in a **readline** initialization file. If arguments remain after option processing, **bind** treats them as **readline** command names and restricts output to those names.

−P List current **readline** function names and bindings. If arguments remain after option processing, **bind** treats them as **readline** command names and restricts output to those names.

−s Display **readline** key sequences bound to macros and the strings they output in such a way that they can be used as an argument to a subsequent **bind** command or in a **readline** initialization file.

−S Display **readline** key sequences bound to macros and the strings they output.

−v Display **readline** variable names and values in such a way that they can be used as an argument to a subsequent **bind** command or in a **readline** initialization file.

−V List current **readline** variable names and values.

−f *filename*
: Read key bindings from *filename*.

−q *function*
: Display key sequences that invoke the named **readline** *function*.

−u *function*
: Unbind all key sequences bound to the named **readline** *function*.

−r *keyseq*
: Remove any current binding for *keyseq*.

−x *keyseq*[**:**]*shell−command*
: Cause *shell−command* to be executed whenever *keyseq* is entered. The separator between *keyseq* and *shell−command* is either whitespace or a colon optionally followed by whitespace. If the separator is whitespace, *shell−command* must be enclosed in double quotes and **readline** expands any of its special backslash-escapes in *shell−command* before saving it. If the separator is a colon, any enclosing double quotes are optional, and **readline** does not expand the command string before saving it. Since the entire key binding expression must be a single argument, it should be enclosed in single quotes. When *shell−command* is executed, the shell sets the **READLINE_LINE** variable to the contents of the **readline** line buffer and the **READLINE_POINT** and **READLINE_MARK** variables to the current location of the insertion point and the saved insertion point (the mark), respectively. The shell assigns any numeric argument the user supplied to the **READLINE_ARGUMENT** variable. If there was no argument, that variable is not set. If the executed command changes the value of any of **READLINE_LINE**, **READLINE_POINT**, or **READLINE_MARK**, those new values will be reflected in the editing state.

−X List all key sequences bound to shell commands and the associated commands in a format that can be reused as an argument to a subsequent **bind** command.

The return value is 0 unless an unrecognized option is supplied or an error occurred.

break [*n*]
: Exit from within a **for**, **while**, **until**, or **select** loop. If *n* is specified, **break** exits *n* enclosing loops. *n* must be ≥ 1. If *n* is greater than the number of enclosing loops, all enclosing loops are exited. The return value is 0 unless *n* is not greater than or equal to 1.

builtin *shell−builtin* [*arguments*]

Execute the specified shell builtin *shell−builtin*, passing it *arguments*, and return its exit status. This is useful when defining a function whose name is the same as a shell builtin, retaining the functionality of the builtin within the function. The **cd** builtin is commonly redefined this way. The return status is false if *shell−builtin* is not a shell builtin command.

caller [*expr*]

Returns the context of any active subroutine call (a shell function or a script executed with the **.** or **source** builtins).

Without *expr*, **caller** displays the line number and source filename of the current subroutine call. If a non-negative integer is supplied as *expr*, **caller** displays the line number, subroutine name, and source file corresponding to that position in the current execution call stack. This extra information may be used, for example, to print a stack trace. The current frame is frame 0.

The return value is 0 unless the shell is not executing a subroutine call or *expr* does not correspond to a valid position in the call stack.

cd [**−L**] [**−@**] [*dir*]
cd −P [**−e**] [**−@**] [*dir*]

Change the current directory to *dir*. if *dir* is not supplied, the value of the **HOME** shell variable is used as *dir*. The variable **CDPATH** exists, and *dir* does not begin with a slash (/), **cd** uses it as a search path: the shell searches each directory name in **CDPATH** for *dir*. Alternative directory names in **CDPATH** are separated by a colon (:). A null directory name in **CDPATH** is the same as the current directory, i.e., ".".

The **−P** option causes **cd** to use the physical directory structure by resolving symbolic links while traversing *dir* and before processing instances of .. in *dir* (see also the **−P** option to the **set** builtin command).

The **−L** option forces **cd** to follow symbolic links by resolving the link after processing instances of .. in *dir*. If .. appears in *dir*, **cd** processes it by removing the immediately previous pathname component from *dir*, back to a slash or the beginning of *dir*, and verifying that the portion of *dir* it has processed to that point is still a valid directory name after removing the pathname component. If it is not a valid directory name, **cd** returns a non-zero status. If neither **−L** nor **−P** is supplied, **cd** behaves as if **−L** had been supplied.

If the **−e** option is supplied with **−P**, and **cd** cannot successfully determine the current working directory after a successful directory change, it returns a non-zero status.

On systems that support it, the **−@** option presents the extended attributes associated with a file as a directory.

An argument of **−** is converted to **$OLDPWD** before attempting the directory change.

If **cd** uses a non-empty directory name from **CDPATH**, or if **−** is the first argument, and the directory change is successful, **cd** writes the absolute pathname of the new working directory to the standard output.

If the directory change is successful, **cd** sets the value of the **PWD** environment variable to the new directory name, and sets the **OLDPWD** environment variable to the value of the current working directory before the change.

The return value is true if the directory was successfully changed; false otherwise.

command [**−pVv**] *command* [*arg* ...]

The **command** builtin runs *command* with *args* suppressing the normal shell function lookup for *command*. Only builtin commands or commands found in the **PATH** named *command* are executed. If the **−p** option is supplied, the search for *command* is performed using a default value for **PATH** that is guaranteed to find all of the standard utilities.

If either the **−V** or **−v** option is supplied, **command** prints a description of *command*. The **−v** option displays a single word indicating the command or filename used to invoke *command*; the **−V** option produces a more verbose description.

If the **−V** or **−v** option is supplied, the exit status is zero if *command* was found, and non-zero if not. If neither option is supplied and an error occurred or *command* cannot be found, the exit status is 127. Otherwise, the exit status of the **command** builtin is the exit status of *command*.

compgen [**−V** *varname*] [*option*] [*word*]

Generate possible completion matches for *word* according to the *option*s, which may be any option accepted by the **complete** builtin with the exceptions of **−p**, **−r**, **−D**, **−E**, and **−I**, and write the matches to the standard output.

If the **−V** option is supplied, **compgen** stores the generated completions into the indexed array variable *varname* instead of writing them to the standard output.

When using the **−F** or **−C** options, the various shell variables set by the programmable completion facilities, while available, will not have useful values.

The matches will be generated in the same way as if the programmable completion code had generated them directly from a completion specification with the same flags. If *word* is specified, only those completions matching *word* will be displayed or stored.

The return value is true unless an invalid option is supplied, or no matches were generated.

complete [**−abcdefgjksuv**] [**−o** *comp-option*] [**−DEI**] [**−A** *action*]
[**−G** *globpat*] [**−W** *wordlist*] [**−F** *function*] [**−C** *command*]
[**−X** *filterpat*] [**−P** *prefix*] [**−S** *suffix*] *name* [*name* ...]
complete −pr [**−DEI**] [*name* ...]

Specify how arguments to each *name* should be completed.

If the **−p** option is supplied, or if no options or *name*s are supplied, print existing completion specifications in a way that allows them to be reused as input. The **−r** option removes a completion specification for each *name*, or, if no *name*s are supplied, all completion specifications.

The **−D** option indicates that other supplied options and actions should apply to the "default" command completion; that is, completion attempted on a command for which no completion has previously been defined. The **−E** option indicates that other supplied options and actions should apply to "empty" command completion; that is, completion attempted on a blank line. The **−I** option indicates that other supplied options and actions should apply to completion on the initial non-assignment word on the line, or after a command delimiter such as **;** or **|**, which is usually command name completion. If multiple options are supplied, the **−D** option takes precedence over **−E**, and both take precedence over **−I**. If any of **−D**, **−E**, or **−I** are supplied, any other *name* arguments are ignored; these completions only apply to the case specified by the option.

The process of applying these completion specifications when attempting word completion is described above under **Programmable Completion**.

Other options, if specified, have the following meanings. The arguments to the **−G**, **−W**, and **−X** options (and, if necessary, the **−P** and **−S** options) should be quoted to protect them from expansion before the **complete** builtin is invoked.

−o *comp-option*

The *comp-option* controls several aspects of the compspec's behavior beyond the simple generation of completions. *comp-option* may be one of:

bashdefault

Perform the rest of the default **bash** completions if the compspec generates no matches.

default Use **readline**'s default filename completion if the compspec generates no matches.

dirnames

Perform directory name completion if the compspec generates no matches.

filenames
Tell **readline** that the compspec generates filenames, so it can perform any filename−specific processing (such as adding a slash to directory names, quoting special characters, or suppressing trailing spaces). This is intended to be used with shell functions.

fullquote
Tell **readline** to quote all the completed words even if they are not filenames.

noquote Tell **readline** not to quote the completed words if they are filenames (quoting filenames is the default).

nosort Tell **readline** not to sort the list of possible completions alphabetically.

nospace Tell **readline** not to append a space (the default) to words completed at the end of the line.

plusdirs After generating any matches defined by the compspec, attempt directory name completion and add any matches to the results of the other actions.

−A *action*
The *action* may be one of the following to generate a list of possible completions:

alias Alias names. May also be specified as **−a**.

arrayvar
Array variable names.

binding **Readline** key binding names.

builtin Names of shell builtin commands. May also be specified as **−b**.

command
Command names. May also be specified as **−c**.

directory
Directory names. May also be specified as **−d**.

disabled
Names of disabled shell builtins.

enabled Names of enabled shell builtins.

export Names of exported shell variables. May also be specified as **−e**.

file File and directory names, similar to **readline**'s filename completion. May also be specified as **−f**.

function
Names of shell functions.

group Group names. May also be specified as **−g**.

helptopic
Help topics as accepted by the **help** builtin.

hostname
Hostnames, as taken from the file specified by the **HOSTFILE** shell variable.

job Job names, if job control is active. May also be specified as **−j**.

keyword
Shell reserved words. May also be specified as **−k**.

running Names of running jobs, if job control is active.

service Service names. May also be specified as **−s**.

setopt Valid arguments for the **−o** option to the **set** builtin.

shopt Shell option names as accepted by the **shopt** builtin.

signal Signal names.

stopped Names of stopped jobs, if job control is active.

user User names. May also be specified as **−u**.

variable Names of all shell variables. May also be specified as **−v**.

−C *command*
command is executed in a subshell environment, and its output is used as the possible completions. Arguments are passed as with the **−F** option.

−F *function*
The shell function *function* is executed in the current shell environment. When the function is executed, the first argument (**$1**) is the name of the command whose arguments are being completed, the second argument (**$2**) is the word being completed, and the third argument (**$3**) is the word preceding the word being completed on the current command line. When *function* finishes, programmable completion retrieves the possible completions from the value of the **COMPREPLY** array variable.

−G *globpat*
Expand the pathname expansion pattern *globpat* to generate the possible completions.

−P *prefix*
Add *prefix* to the beginning of each possible completion after all other options have been applied.

−S *suffix* Append *suffix* to each possible completion after all other options have been applied.

−W *wordlist*
Split the *wordlist* using the characters in the **IFS** special variable as delimiters, and expand each resulting word. Shell quoting is honored within *wordlist*, in order to provide a mechanism for the words to contain shell metacharacters or characters in the value of **IFS**. The possible completions are the members of the resultant list which match a prefix of the word being completed.

−X *filterpat*
filterpat is a pattern as used for pathname expansion. It is applied to the list of possible completions generated by the preceding options and arguments, and each completion matching *filterpat* is removed from the list. A leading **!** in *filterpat* negates the pattern; in this case, any completion not matching *filterpat* is removed.

The return value is true unless an invalid option is supplied, an option other than **−p**, **−r**, **−D**, **−E**, or **−I** is supplied without a *name* argument, an attempt is made to remove a completion specification for a *name* for which no specification exists, or an error occurs adding a completion specification.

compopt [**−o** *option*] [**−DEI**] [**+o** *option*] [*name*]
Modify completion options for each *name* according to the *option*s, or for the currently-executing completion if no *name*s are supplied. If no *option*s are supplied, display the completion options for each *name* or the current completion. The possible values of *option* are those valid for the **complete** builtin described above.

The **−D** option indicates that other supplied options should apply to the “default” command completion; the **−E** option indicates that other supplied options should apply to “empty” command completion; and the **−I** option indicates that other supplied options should apply to completion on the initial word on the line. These are determined in the same way as the **complete** builtin.

If multiple options are supplied, the **−D** option takes precedence over **−E**, and both take precedence over **−I**.

The return value is true unless an invalid option is supplied, an attempt is made to modify the options for a *name* for which no completion specification exists, or an output error occurs.

continue [*n*]
continue resumes the next iteration of the enclosing **for**, **while**, **until**, or **select** loop. If *n* is specified, **bash** resumes the *n*th enclosing loop. *n* must be ≥ 1. If *n* is greater than the number of enclosing loops, the shell resumes the last enclosing loop (the “top-level” loop). The return value is 0 unless *n* is not greater than or equal to 1.

declare [**−aAfFgiIlnrtux**] [**−p**] [*name*[=*value*] ...]
typeset [**−aAfFgiIlnrtux**] [**−p**] [*name*[=*value*] ...]
Declare variables and/or give them attributes. If no *name*s are given then display the values of variables or functions. The **−p** option will display the attributes and values of each *name*. When **−p** is used with *name* arguments, additional options, other than **−f** and **−F**, are ignored.

When **−p** is supplied without *name* arguments, **declare** will display the attributes and values of all variables having the attributes specified by the additional options. If no other options are supplied with **−p**, **declare** will display the attributes and values of all shell variables. The **−f** option restricts the display to shell functions.

The **−F** option inhibits the display of function definitions; only the function name and attributes are printed. If the **extdebug** shell option is enabled using **shopt**, the source file name and line number where each *name* is defined are displayed as well. The **−F** option implies **−f**.

The **−g** option forces variables to be created or modified at the global scope, even when **declare** is executed in a shell function. It is ignored when **declare** is not executed in a shell function.

The **−I** option causes local variables to inherit the attributes (except the *nameref* attribute) and value of any existing variable with the same *name* at a surrounding scope. If there is no existing variable, the local variable is initially unset.

The following options can be used to restrict output to variables with the specified attribute or to give variables attributes:

- **−a** Each *name* is an indexed array variable (see **Arrays** above).
- **−A** Each *name* is an associative array variable (see **Arrays** above).
- **−f** Each *name* refers to a shell function.
- **−i** The variable is treated as an integer; arithmetic evaluation (see **ARITHMETIC EVALUATION** above) is performed when the variable is assigned a value.
- **−l** When the variable is assigned a value, all upper-case characters are converted to lower-case. The upper-case attribute is disabled.
- **−n** Give each *name* the *nameref* attribute, making it a name reference to another variable. That other variable is defined by the value of *name*. All references, assignments, and attribute modifications to *name*, except those using or changing the **−n** attribute itself, are performed on the variable referenced by *name*'s value. The nameref attribute cannot be applied to array variables.
- **−r** Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements or unset.
- **−t** Give each *name* the *trace* attribute. Traced functions inherit the **DEBUG** and **RETURN** traps from the calling shell. The trace attribute has no special meaning for variables.
- **−u** When the variable is assigned a value, all lower-case characters are converted to upper-case. The lower-case attribute is disabled.
- **−x** Mark each *name* for export to subsequent commands via the environment.

Using "+" instead of "−" turns off the specified attribute instead, with the exceptions that **+a** and **+A** may not be used to destroy array variables and **+r** will not remove the readonly attribute.

When used in a function, **declare** and **typeset** make each *name* local, as with the **local** command, unless the **−g** option is supplied. If a variable name is followed by =*value*, the value of the variable is set to *value*. When using **−a** or **−A** and the compound assignment syntax to create array variables, additional attributes do not take effect until subsequent assignments.

The return value is 0 unless an invalid option is encountered, an attempt is made to define a function using "−f foo=bar", an attempt is made to assign a value to a readonly variable, an attempt is made to assign a value to an array variable without using the compound assignment syntax (see **Arrays** above), one of the *names* is not a valid shell variable name, an attempt is made to turn off readonly status for a readonly variable, an attempt is made to turn off array status for an array variable, or an attempt is made to display a non-existent function with **−f**.

dirs [−clpv] [+*n*] [−*n*]

Without options, display the list of currently remembered directories. The default display is on a single line with directory names separated by spaces. Directories are added to the list with the **pushd** command; the **popd** command removes entries from the list. The current directory is always the first directory in the stack.

Options, if supplied, have the following meanings:

−c Clears the directory stack by deleting all of the entries.

−l Produces a listing using full pathnames; the default listing format uses a tilde to denote the home directory.

−p Print the directory stack with one entry per line.

−v Print the directory stack with one entry per line, prefixing each entry with its index in the stack.

+*n* Displays the *n*th entry counting from the left of the list shown by **dirs** when invoked without options, starting with zero.

−*n* Displays the *n*th entry counting from the right of the list shown by **dirs** when invoked without options, starting with zero.

The return value is 0 unless an invalid option is supplied or *n* indexes beyond the end of the directory stack.

disown [**−ar**] [**−h**] [*id* ...]

Without options, remove each *id* from the table of active jobs. Each *id* may be a job specification *jobspec* or a process ID *pid*; if *id* is a *pid*, **disown** uses the job containing *pid* as *jobspec*.

If the **−h** option is supplied, **disown** does not remove the jobs corresponding to each *id* from the jobs table, but rather marks them so the shell does not send **SIGHUP** to the job if the shell receives a **SIGHUP**.

If no *id* is supplied, the **−a** option means to remove or mark all jobs; the **−r** option without an *id* argument removes or marks running jobs. If no *id* is supplied, and neither the **−a** nor the **−r** option is supplied, **disown** removes or marks the current job.

The return value is 0 unless an *id* does not specify a valid job.

echo [**−neE**] [*arg* ...]

Output the *arg*s, separated by spaces, followed by a newline. The return status is 0 unless a write error occurs. If **−n** is specified, the trailing newline is not printed.

If the **−e** option is given, **echo** interprets the following backslash-escaped characters. The **−E** option disables interpretation of these escape characters, even on systems where they are interpreted by default. The **xpg_echo** shell option determines whether or not **echo** interprets any options and expands these escape characters. **echo** does not interpret **−−** to mean the end of options.

echo interprets the following escape sequences:

\a alert (bell)

\b backspace

\c suppress further output

\e

\E an escape character

\f form feed

\n new line

\r carriage return

\t horizontal tab

\v vertical tab

**** backslash

\0*nnn* The eight-bit character whose value is the octal value *nnn* (zero to three octal digits).

\x*HH* The eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits).

\u*HHHH*

The Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHH* (one to four hex digits).

\U*HHHHHHHH*

The Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHHHHHH* (one to eight hex digits).

echo writes any unrecognized backslash-escaped characters unchanged.

enable [**−a**] [**−dnps**] [**−f** *filename*] [*name* ...]

Enable and disable builtin shell commands. Disabling a builtin allows an executable file which has the same name as a shell builtin to be executed without specifying a full pathname, even though the shell normally searches for builtins before files.

If **−n** is supplied, each *name* is disabled; otherwise, *name*s are enabled. For example, to use the **test** binary found using **PATH** instead of the shell builtin version, run "enable −n test".

If no *name* arguments are supplied, or if the **−p** option is supplied, print a list of shell builtins. With no other option arguments, the list consists of all enabled shell builtins. If **−n** is supplied, print only disabled builtins. If **−a** is supplied, the list printed includes all builtins, with an indication of whether or not each is enabled. The **−s** option means to restrict the output to the POSIX *special* builtins.

The **−f** option means to load the new builtin command *name* from shared object *filename*, on systems that support dynamic loading. If *filename* does not contain a slash, **Bash** will use the value of the **BASH_LOADABLES_PATH** variable as a colon-separated list of directories in which to search for *filename*. The default for **BASH_LOADABLES_PATH** is system-dependent, and may include "." to force a search of the current directory. The **−d** option will delete a builtin previously loaded with **−f**. If *−s* is used with *−f*, the new builtin becomes a POSIX special builtin.

If no options are supplied and a *name* is not a shell builtin, **enable** will attempt to load *name* from a shared object named *name*, as if the command were "enable −f *name name*".

The return value is 0 unless a *name* is not a shell builtin or there is an error loading a new builtin from a shared object.

eval [*arg* ...]

Concatenate the *arg*s together into a single command, separating them with spaces. **Bash** then reads and execute this command, and returns its exit status as the return status of **eval**. If there are no *args*, or only null arguments, **eval** returns 0.

exec [**−cl**] [**−a** *name*] [*command* [*arguments*]]

If *command* is specified, it replaces the shell without creating a new process. *command* cannot be a shell builtin or function. The *arguments* become the arguments to *command*. If the **−l** option is supplied, the shell places a dash at the beginning of the zeroth argument passed to *command*. This is what *login*(1) does. The **−c** option causes *command* to be executed with an empty environment. If **−a** is supplied, the shell passes *name* as the zeroth argument to the executed command.

If *command* cannot be executed for some reason, a non-interactive shell exits, unless the **execfail** shell option is enabled. In that case, it returns a non-zero status. An interactive shell returns a non-zero status if the file cannot be executed. A subshell exits unconditionally if **exec** fails.

If *command* is not specified, any redirections take effect in the current shell, and the return status is 0. If there is a redirection error, the return status is 1.

exit [*n*]

Cause the shell to exit with a status of *n*. If *n* is omitted, the exit status is that of the last command executed. Any trap on **EXIT** is executed before the shell terminates.

export [**−fn**] [*name*[=*value*]] ...
export −p [−f]

The supplied *names* are marked for automatic export to the environment of subsequently executed commands. If the **−f** option is given, the *names* refer to functions.

The **−n** option unexports, or removes the export attribute, from each *name*. If no *names* are given, or if only the **−p** option is supplied, **export** displays a list of names of all exported variables on the standard output. Using **−p** and **−f** together displays exported functions. The **−p** option displays output in a form that may be reused as input.

export allows the value of a variable to be set when it is exported or unexported by following the variable name with =*value*. This sets the value of the variable to *value* while modifying the export attribute. **export** returns an exit status of 0 unless an invalid option is encountered, one of the *names* is not a valid shell variable name, or **−f** is supplied with a *name* that is not a function.

false Does nothing; returns a non-zero status.

fc [**−e** *ename*] [**−lnr**] [*first*] [*last*]
fc −s [*pat*=*rep*] [*cmd*]

The first form selects a range of commands from *first* to *last* from the history list and displays or edits and re-executes them. *First* and *last* may be specified as a string (to locate the last command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number).

When listing, a *first* or *last* of 0 is equivalent to −1 and −0 is equivalent to the current command (usually the **fc** command); otherwise 0 is equivalent to −1 and −0 is invalid. If *last* is not specified, it is set to the current command for listing (so that "fc −l −10" prints the last 10 commands) and to *first* otherwise. If *first* is not specified, it is set to the previous command for editing and −16 for listing.

If the **−l** option is supplied, the commands are listed on the standard output. The **−n** option suppresses the command numbers when listing. The **−r** option reverses the order of the commands.

Otherwise, **fc** invokes the editor named by *ename* on a file containing those commands. If *ename* is not supplied, **fc** uses the value of the **FCEDIT** variable, and the value of **EDITOR** if **FCEDIT** is not set. If neither variable is set, **fc** uses *vi.* When editing is complete, **fc** reads the file containing the edited commands and echoes and executes them.

In the second form, **fc** re-executes *command* after replacing each instance of *pat* with *rep*. *Command* is interpreted the same as *first* above.

A useful alias to use with **fc** is "r="fc −s"", so that typing "r cc" runs the last command beginning with "cc" and typing "r" re-executes the last command.

If the first form is used, the return value is zero unless an invalid option is encountered or *first* or *last* specify history lines out of range. When editing and re-executing a file of commands, the return value is the value of the last command executed or failure if an error occurs with the temporary file. If the second form is used, the return status is that of the re-executed command, unless *cmd* does not specify a valid history entry, in which case **fc** returns a non-zero status.

fg [*jobspec*]

Resume *jobspec* in the foreground, and make it the current job. If *jobspec* is not present, **fg** uses the shell's notion of the *current job*. The return value is that of the command placed into the foreground, or failure if run when job control is disabled or, when run with job control enabled, if *jobspec* does not specify a valid job or *jobspec* specifies a job that was started without job control.

getopts *optstring name* [*arg* ...]

getopts is used by shell scripts and functions to parse positional parameters and obtain options and their arguments. *optstring* contains the option characters to be recognized; if a character is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. The colon and question mark characters may not be used as option characters.

Each time it is invoked, **getopts** places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable **OPTIND**. **OPTIND** is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, **getopts** places that argument into the variable **OPTARG**.

The shell does not reset **OPTIND** automatically; it must be manually reset between multiple calls to **getopts** within the same shell invocation to use a new set of parameters.

When it reaches the end of options, **getopts** exits with a return value greater than zero. **OPTIND** is set to the index of the first non-option argument, and *name* is set to ?.

getopts normally parses the positional parameters, but if more arguments are supplied as *arg* values, **getopts** parses those instead.

getopts can report errors in two ways. If the first character of *optstring* is a colon, **getopts** uses *silent* error reporting. In normal operation, **getopts** prints diagnostic messages when it encounters invalid options or missing option arguments. If the variable **OPTERR** is set to 0, **getopts** does not display any error messages, even if the first character of *optstring* is not a colon.

If **getopts** detects an invalid option, it places ? into *name* and, if not silent, prints an error message and unsets **OPTARG**. If **getopts** is silent, it assigns the option character found to **OPTARG** and does not print a diagnostic message.

If a required argument is not found, and **getopts** is not silent, it sets the value of *name* to a question mark (**?**), unsets **OPTARG**, and prints a diagnostic message. If **getopts** is silent, it sets the value of *name* to a colon (**:**) and sets **OPTARG** to the option character found.

getopts returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

hash [**−lr**] [**−p** *filename*] [**−dt**] [*name*]

Each time **hash** is invoked, it remembers the full pathname of the command *name* as determined by searching the directories in **$PATH**. Any previously-remembered pathname associated with *name* is discarded. If the **−p** option is supplied, **hash** uses *filename* as the full pathname of the command.

The **−r** option causes the shell to forget all remembered locations. Assigning to the **PATH** variable also clears all hashed filenames. The **−d** option causes the shell to forget the remembered location of each *name*.

If the **−t** option is supplied, **hash** prints the full pathname corresponding to each *name*. If multiple *name* arguments are supplied with **−t**, **hash** prints the *name* before the corresponding hashed full pathname. The **−l** option displays output in a format that may be reused as input.

If no arguments are given, or if only **−l** is supplied, **hash** prints information about remembered commands. The **−t**, **−d**, and **−p** options (the options that act on the *name* arguments) are mutually exclusive. Only one will be active. If more than one is supplied, **−t** has higher priority than **−p**, and both have higher priority than **−d**.

The return status is zero unless a *name* is not found or an invalid option is supplied.

help [**−dms**] [*pattern*]

Display helpful information about builtin commands. If *pattern* is specified, **help** gives detailed help on all commands matching *pattern* as described below; otherwise it displays a list of all the builtins and shell compound commands.

Options, if supplied, have the follow meanings:

−d Display a short description of each *pattern*
−m Display the description of each *pattern* in a manpage-like format
−s Display only a short usage synopsis for each *pattern*

If *pattern* contains pattern matching characters (see **Pattern Matching** above) it's treated as a shell pattern and **help** prints the description of each help topic matching *pattern*.

If not, and *pattern* exactly matches the name of a help topic, **help** prints the description associated with that topic. Otherwise, **help** performs prefix matching and prints the descriptions of all matching help topics.

The return status is 0 unless no command matches *pattern*.

history [*n***]**
history −c
history −d *offset*
history −d *start-end*

history −anrw [*filename*]
history −p *arg* [*arg* . . .]
history −s *arg* [*arg* . . .]

With no options, display the command history list with numbers. Entries prefixed with a * have been modified. An argument of *n* lists only the last *n* entries. If the shell variable **HISTTIMEFORMAT** is set and not null, it is used as a format string for *strftime*(3) to display the time stamp associated with each displayed history entry. If **history** uses **HISTTIMEFORMAT**, it does not print an intervening space between the formatted time stamp and the history entry.

If *filename* is supplied, **history** uses it as the name of the history file; if not, it uses the value of **HISTFILE**. If *filename* is not supplied and **HISTFILE** is unset or null, the **−a, −n, −r,** and **−w** options have no effect.

Options, if supplied, have the following meanings:

−c Clear the history list by deleting all the entries. This can be used with the other options to replace the history list.

−d *offset*
Delete the history entry at position *offset*. If *offset* is negative, it is interpreted as relative to one greater than the last history position, so negative indices count back from the end of the history, and an index of −1 refers to the current **history −d** command.

−d *start*−*end*
Delete the range of history entries between positions *start* and *end*, inclusive. Positive and negative values for *start* and *end* are interpreted as described above.

−a Append the "new" history lines to the history file. These are history lines entered since the beginning of the current **bash** session, but not already appended to the history file.

−n Read the history lines not already read from the history file and add them to the current history list. These are lines appended to the history file since the beginning of the current **bash** session.

−r Read the history file and append its contents to the current history list.

−w Write the current history list to the history file, overwriting the history file.

−p Perform history substitution on the following *args* and display the result on the standard output, without storing the results in the history list. Each *arg* must be quoted to disable normal history expansion.

−s Store the *args* in the history list as a single entry. The last command in the history list is removed before adding the *args*.

If the **HISTTIMEFORMAT** variable is set, **history** writes the time stamp information associated with each history entry to the history file, marked with the history comment character as described above. When the history file is read, lines beginning with the history comment character followed immediately by a digit are interpreted as timestamps for the following history entry.

The return value is 0 unless an invalid option is encountered, an error occurs while reading or writing the history file, an invalid *offset* or range is supplied as an argument to **−d**, or the history expansion supplied as an argument to **−p** fails.

jobs [**−lnprs**] [*jobspec* . . .]
jobs −x *command* [*args* . . .]

The first form lists the active jobs. The options have the following meanings:

−l List process IDs in addition to the normal information.

−n Display information only about jobs that have changed status since the user was last notified of their status.

−p List only the process ID of the job's process group leader.

−r Display only running jobs.

−s Display only stopped jobs.

If *jobspec* is supplied, **jobs** restricts output to information about that job. The return status is 0 unless an invalid option is encountered or an invalid *jobspec* is supplied.

If the **−x** option is supplied, **jobs** replaces any *jobspec* found in *command* or *args* with the corresponding process group ID, and executes *command*, passing it *args*, returning its exit status.

kill [**−s** *sigspec* | **−n** *signum* | **−***sigspec*] *id* [. . .]
kill −l|**−L** [*sigspec* | *exit_status*]

Send the signal specified by *sigspec* or *signum* to the processes named by each *id*. Each *id* may be a job specification *jobspec* or a process ID *pid*. *sigspec* is either a case-insensitive signal name such as **SIGKILL** (with or without the **SIG** prefix) or a signal number; *signum* is a signal number. If *sigspec* is not supplied, then **kill** sends **SIGTERM**.

The **−l** option lists the signal names. If any arguments are supplied when **−l** is given, **kill** lists the names of the signals corresponding to the arguments, and the return status is 0. The *exit_status* argument to **−l** is a number specifying either a signal number or the exit status of a process terminated by a signal; if it is supplied, **kill** prints the name of the signal that caused the process to terminate. **kill** assumes that process exit statuses are greater than 128; anything less than that is a signal number. The **−L** option is equivalent to **−l**.

kill returns true if at least one signal was successfully sent, or false if an error occurs or an invalid option is encountered.

let *arg* [*arg* . . .]

Each *arg* is evaluated as an arithmetic expression (see **ARITHMETIC EVALUATION** above). If the last *arg* evaluates to 0, **let** returns 1; otherwise **let** returns 0.

local [*option*] [*name*[=*value*] . . . | −]

For each argument, create a local variable named *name* and assign it *value*. The *option* can be any of the options accepted by **declare**. When **local** is used within a function, it causes the variable *name* to have a visible scope restricted to that function and its children. It is an error to use **local** when not within a function.

If *name* is −, it makes the set of shell options local to the function in which **local** is invoked: any shell options changed using the **set** builtin inside the function after the call to **local** are restored to their original values when the function returns. The restore is performed as if a series of **set** commands were executed to restore the values that were in place before the function.

With no operands, **local** writes a list of local variables to the standard output.

The return status is 0 unless **local** is used outside a function, an invalid *name* is supplied, or *name* is a readonly variable.

logout [*n***]**

Exit a login shell, returning a status of *n* to the shell's parent.

mapfile [**−d** *delim*] [**−n** *count*] [**−O** *origin*] [**−s** *count*] [**−t**] [**−u** *fd*] [**−C** *callback*] [**−c** *quantum*] [*array*]
readarray [**−d** *delim*] [**−n** *count*] [**−O** *origin*] [**−s** *count*] [**−t**] [**−u** *fd*] [**−C** *callback*] [**−c** *quantum*] [*array*]

Read lines from the standard input, or from file descriptor *fd* if the **−u** option is supplied, into the indexed array variable *array*. The variable **MAPFILE** is the default *array*. Options, if supplied, have the following meanings:

- **−d** Use the first character of *delim* to terminate each input line, rather than newline. If *delim* is the empty string, **mapfile** will terminate a line when it reads a NUL character.
- **−n** Copy at most *count* lines. If *count* is 0, copy all lines.
- **−O** Begin assigning to *array* at index *origin*. The default index is 0.
- **−s** Discard the first *count* lines read.
- **−t** Remove a trailing *delim* (default newline) from each line read.
- **−u** Read lines from file descriptor *fd* instead of the standard input.
- **−C** Evaluate *callback* each time *quantum* lines are read. The **−c** option specifies *quantum*.
- **−c** Specify the number of lines read between each call to *callback*.

If **−C** is specified without **−c**, the default quantum is 5000. When *callback* is evaluated, it is supplied the index of the next array element to be assigned and the line to be assigned to that element as additional arguments. *callback* is evaluated after the line is read but before the array element is assigned.

If not supplied with an explicit origin, **mapfile** will clear *array* before assigning to it.

mapfile returns zero unless an invalid option or option argument is supplied, *array* is invalid or unassignable, or if *array* is not an indexed array.

popd [**−n**] [+*n*] [−*n*]

Remove entries from the directory stack. The elements are numbered from 0 starting at the first directory listed by **dirs**, so **popd** is equivalent to "popd +0." With no arguments, **popd** removes the top directory from the stack, and changes to the new top directory. Arguments, if supplied, have the following meanings:

−n Suppress the normal change of directory when removing directories from the stack, only manipulate the stack.

+*n* Remove the *n*th entry counting from the left of the list shown by **dirs**, starting with zero, from the stack. For example: "popd +0" removes the first directory, "popd +1" the second.

−*n* Remove the *n*th entry counting from the right of the list shown by **dirs**, starting with zero. For example: "popd −0" removes the last directory, "popd −1" the next to last.

If the top element of the directory stack is modified, and the −*n* option was not supplied, **popd** uses the **cd** builtin to change to the directory at the top of the stack. If the **cd** fails, **popd** returns a non-zero value.

Otherwise, **popd** returns false if an invalid option is supplied, the directory stack is empty, or *n* specifies a non-existent directory stack entry.

If the **popd** command is successful, **bash** runs **dirs** to show the final contents of the directory stack, and the return status is 0.

printf [**−v** *var*] *format* [*arguments*]

Write the formatted *arguments* to the standard output under the control of the *format*. The **−v** option assigns the output to the variable *var* rather than printing it to the standard output.

The *format* is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences, which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive *argument*. In addition to the standard *printf*(3) format characters **cCsSndiouxXeEfFgGaA**, **printf** interprets the following additional format specifiers:

%b causes **printf** to expand backslash escape sequences in the corresponding *argument* in the same way as **echo −e**.

%q causes **printf** to output the corresponding *argument* in a format that can be reused as shell input. **%q** and **%Q** use the **$''** quoting style if any characters in the argument string require it, and backslash quoting otherwise. If the format string uses the *printf* alternate form, these two formats quote the argument string using single quotes.

%Q like **%q**, but applies any supplied precision to the *argument* before quoting it.

%(*datefmt***)T**

causes **printf** to output the date-time string resulting from using *datefmt* as a format string for *strftime*(3). The corresponding *argument* is an integer representing the number of seconds since the epoch. This format specifier recognizes two special argument values: −1 represents the current time, and −2 represents the time the shell was invoked. If no argument is specified, conversion behaves as if −1 had been supplied. This is an exception to the usual **printf** behavior.

The %b, %q, and %T format specifiers all use the field width and precision arguments from the format specification and write that many bytes from (or use that wide a field for) the expanded argument, which usually contains more characters than the original.

The %n format specifier accepts a corresponding argument that is treated as a shell variable name.

The %s and %c format specifiers accept an l (long) modifier, which forces them to convert the argument string to a wide-character string and apply any supplied field width and precision in terms of characters, not bytes. The %S and %C format specifiers are equivalent to %ls and %lc, respectively.

Arguments to non-string format specifiers are treated as C constants, except that a leading plus or minus sign is allowed, and if the leading character is a single or double quote, the value is the numeric value of the following character, using the current locale.

The *format* is reused as necessary to consume all of the *arguments*. If the *format* requires more *arguments* than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied. The return value is zero on success, non-zero if an invalid option is supplied or a write or assignment error occurs.

pushd [**−n**] [+*n*] [−*n*]
pushd [**−n**] [*dir*]

Add a directory to the top of the directory stack, or rotate the stack, making the new top of the stack the current working directory. With no arguments, **pushd** exchanges the top two elements of the directory stack. Arguments, if supplied, have the following meanings:

−n Suppress the normal change of directory when rotating or adding directories to the stack, only manipulate the stack.

+*n* Rotate the stack so that the *n*th directory (counting from the left of the list shown by **dirs**, starting with zero) is at the top.

−*n* Rotates the stack so that the *n*th directory (counting from the right of the list shown by **dirs**, starting with zero) is at the top.

dir Adds *dir* to the directory stack at the top.

After the stack has been modified, if the **−n** option was not supplied, **pushd** uses the **cd** builtin to change to the directory at the top of the stack. If the **cd** fails, **pushd** returns a non-zero value.

Otherwise, if no arguments are supplied, **pushd** returns zero unless the directory stack is empty. When rotating the directory stack, **pushd** returns zero unless the directory stack is empty or *n* specifies a non-existent directory stack element.

If the **pushd** command is successful, **bash** runs **dirs** to show the final contents of the directory stack.

pwd [**−LP**]

Print the absolute pathname of the current working directory. The pathname printed contains no symbolic links if the **−P** option is supplied or the **−o physical** option to the **set** builtin command is enabled. If the **−L** option is used, the pathname printed may contain symbolic links. The return status is 0 unless an error occurs while reading the name of the current directory or an invalid option is supplied.

read [**−Eers**] [**−a** *aname*] [**−d** *delim*] [**−i** *text*] [**−n** *nchars*] [**−N** *nchars*] [**−p** *prompt*] [**−t** *timeout*] [**−u** *fd*] [*name* ...]

Read one line from the standard input, or from the file descriptor *fd* supplied as an argument to the **−u** option, split it into words as described above under **Word Splitting**, and assign the first word to the first *name*, the second word to the second *name*, and so on. If there are more words than names, the remaining words and their intervening delimiters are assigned to the last *name*. If there are fewer words read from the input stream than names, the remaining names are assigned empty values. The characters in the value of the **IFS** variable are used to split the line into words using the same rules the shell uses for expansion (described above under **Word Splitting**). The backslash character (****) removes any special meaning for the next character read and is used for line continuation.

Options, if supplied, have the following meanings:

−a *aname*

The words are assigned to sequential indices of the array variable *aname*, starting at 0. *aname* is unset before any new values are assigned. Other *name* arguments are ignored.

−d *delim*
The first character of *delim* terminates the input line, rather than newline. If *delim* is the empty string, **read** will terminate a line when it reads a NUL character.

−e If the standard input is coming from a terminal, **read** uses **readline** (see **READLINE** above) to obtain the line. **Readline** uses the current (or default, if line editing was not previously active) editing settings, but uses **readline**'s default filename completion.

−E If the standard input is coming from a terminal, **read** uses **readline** (see **READLINE** above) to obtain the line. **Readline** uses the current (or default, if line editing was not previously active) editing settings, but uses bash's default completion, including programmable completion.

−i *text* If **readline** is being used to read the line, **read** places *text* into the editing buffer before editing begins.

−n *nchars*
read returns after reading *nchars* characters rather than waiting for a complete line of input, unless it encounters EOF or **read** times out, but honors a delimiter if it reads fewer than *nchars* characters before the delimiter.

−N *nchars*
read returns after reading exactly *nchars* characters rather than waiting for a complete line of input, unless it encounters EOF or **read** times out. Any delimiter characters in the input are not treated specially and do not cause **read** to return until it has read *nchars* characters. The result is not split on the characters in **IFS**; the intent is that the variable is assigned exactly the characters read (with the exception of backslash; see the **−r** option below).

−p *prompt*
Display *prompt* on standard error, without a trailing newline, before attempting to read any input, but only if input is coming from a terminal.

−r Backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not then be used as a line continuation.

−s Silent mode. If input is coming from a terminal, characters are not echoed.

−t *timeout*
Cause **read** to time out and return failure if it does not read a complete line of input (or a specified number of characters) within *timeout* seconds. *timeout* may be a decimal number with a fractional portion following the decimal point. This option is only effective if **read** is reading input from a terminal, pipe, or other special file; it has no effect when reading from regular files. If **read** times out, it saves any partial input read into the specified variable *name*, and the exit status is greater than 128. If *timeout* is 0, **read** returns immediately, without trying to read any data. In this case, the exit status is 0 if input is available on the specified file descriptor, or the read will return EOF, non-zero otherwise.

−u *fd* Read input from file descriptor *fd* instead of the standard input.

Other than the case where *delim* is the empty string, **read** ignores any NUL characters in the input.

If no *names* are supplied, **read** assigns the line read, without the ending delimiter but otherwise unmodified, to the variable **REPLY**.

The exit status is zero, unless end-of-file is encountered, **read** times out (in which case the status is greater than 128), a variable assignment error (such as assigning to a readonly variable) occurs, or an invalid file descriptor is supplied as the argument to **−u**.

readonly [**−aAf**] [**−p**] [*name*[=*word*] . . .]
The given *names* are marked readonly; the values of these *names* may not be changed by subsequent assignment or unset. If the **−f** option is supplied, each *name* refers to a shell function. The **−a** option restricts the variables to indexed arrays; the **−A** option restricts the variables to associative arrays. If both options are supplied, **−A** takes precedence. If no *name* arguments are supplied, or if the **−p** option is supplied, print a list of all readonly names. The other options may be used to restrict the output to a subset of the set of readonly names. The **−p** option displays output

in a format that may be reused as input.

readonly allows the value of a variable to be set at the same time the readonly attribute is changed by following the variable name with =*value*. This sets the value of the variable is to *value* while modifying the readonly attribute.

The return status is 0 unless an invalid option is encountered, one of the *names* is not a valid shell variable name, or **−f** is supplied with a *name* that is not a function.

return [*n*]

Stop executing a shell function or sourced file and return the value specified by *n* to its caller. If *n* is omitted, the return status is that of the last command executed. If **return** is executed by a trap handler, the last command used to determine the status is the last command executed before the trap handler. If **return** is executed during a **DEBUG** trap, the last command used to determine the status is the last command executed by the trap handler before **return** was invoked.

When **return** is used to terminate execution of a script being executed by the **.** (**source**) command, it causes the shell to stop executing that script and return either *n* or the exit status of the last command executed within the script as the exit status of the script. If *n* is supplied, the return value is its least significant 8 bits.

Any command associated with the **RETURN** trap is executed before execution resumes after the function or script.

The return status is non-zero if **return** is supplied a non-numeric argument, or is used outside a function and not during execution of a script by **.** or **source**.

set [**−abefhkmnptuvxBCEHPT**] [**−o** *option−name*] [**−−**] [**−**] [*arg* ...]
set [**+abefhkmnptuvxBCEHPT**] [**+o** *option−name*] [**−−**] [**−**] [*arg* ...]
set −o
set +o

Without options, display the name and value of each shell variable in a format that can be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In posix mode, only shell variables are listed. The output is sorted according to the current locale. When options are specified, they set or unset shell attributes. Any arguments remaining after option processing are treated as values for the positional parameters and are assigned, in order, to **$1**, **$2**, ..., **$***n*. Options, if specified, have the following meanings:

−a Each variable or function that is created or modified is given the export attribute and marked for export to the environment of subsequent commands.

−b Report the status of terminated background jobs immediately, rather than before the next primary prompt or after a foreground command terminates. This is effective only when job control is enabled.

−e Exit immediately if a *pipeline* (which may consist of a single *simple command*), a *list*, or a *compound command* (see **SHELL GRAMMAR** above), exits with a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a **while** or **until** reserved word, part of the test following the **if** or **elif** reserved words, part of any command executed in a **&&** or **||** list except the command following the final **&&** or **||**, any command in a pipeline but the last (subject to the state of the **pipefail** shell option), or if the command's return value is being inverted with **!**. If a compound command other than a subshell returns a non-zero status because a command failed while **−e** was being ignored, the shell does not exit. A trap on **ERR**, if set, is executed before the shell exits. This option applies to the shell environment and each subshell environment separately (see **COMMAND EXECUTION ENVIRONMENT** above), and may cause subshells to exit before executing all the commands in the subshell.

If a compound command or shell function executes in a context where **−e** is being ignored, none of the commands executed within the compound command or function body will be affected by the **−e** setting, even if **−e** is set and a command returns a failure status. If a compound command or shell function sets **−e** while executing in a context where **−e** is ignored, that setting will not have any effect until the compound command or the command containing the function call completes.

−f Disable pathname expansion.

−h Remember the location of commands as they are looked up for execution. This is enabled by default.

−k All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.

−m Monitor mode. Job control is enabled. This option is on by default for interactive shells on systems that support it (see **JOB CONTROL** above). All processes run in a separate process group. When a background job completes, the shell prints a line containing its exit status.

−n Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored by interactive shells.

−o *option−name*

The *option−name* can be one of the following:

allexport
Same as **−a**.

braceexpand
Same as **−B**.

emacs Use an emacs-style command line editing interface. This is enabled by default when the shell is interactive, unless the shell is started with the **−−noediting** option. This also affects the editing interface used for **read −e**.

errexit Same as **−e**.

errtrace Same as **−E**.

functrace
Same as **−T**.

hashall Same as **−h**.

histexpand
Same as **−H**.

history Enable command history, as described above under **HISTORY**. This option is on by default in interactive shells.

ignoreeof
The effect is as if the shell command "IGNOREEOF=10" had been executed (see **Shell Variables** above).

keyword
Same as **−k**.

monitor Same as **−m**.

noclobber
Same as **−C**.

noexec Same as **−n**.

noglob Same as **−f**.

nolog Currently ignored.

notify Same as **−b**.

nounset Same as **−u**.

onecmd Same as **−t**.

physical Same as **−P**.

pipefail If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

posix Enable posix mode; change the behavior of **bash** where the default operation differs from the POSIX standard to match the standard. See **SEE ALSO** below for a reference to a document that details how posix mode affects bash's behavior.

privileged
Same as **−p**.

verbose Same as **−v**.

vi Use a vi-style command line editing interface. This also affects the editing interface used for **read −e**.

xtrace Same as **−x**.

If **−o** is supplied with no *option−name*, **set** prints the current shell option settings. If **+o** is supplied with no *option−name*, **set** prints a series of **set** commands to recreate the current option settings on the standard output.

−p Turn on *privileged* mode. In this mode, the shell does not read the **$ENV** and **$BASH_ENV** files, shell functions are not inherited from the environment, and the **SHELLOPTS**, **BASHOPTS**, **CDPATH**, and **GLOBIGNORE** variables, if they appear in the environment, are ignored. If the shell is started with the effective user (group) id not equal to the real user (group) id, and the **−p** option is not supplied, these actions are taken and the effective user id is set to the real user id. If the **−p** option is supplied at startup, the effective user id is not reset. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

−r Enable restricted shell mode. This option cannot be unset once it has been set.

−t Exit after reading and executing one command.

−u Treat unset variables and parameters other than the special parameters "@" and "*", or array variables subscripted with "@" or "*", as an error when performing parameter expansion. If expansion is attempted on an unset variable or parameter, the shell prints an error message, and, if not interactive, exits with a non-zero status.

−v Print shell input lines as they are read.

−x After expanding each *simple command*, **for** command, **case** command, **select** command, or arithmetic **for** command, display the expanded value of **PS4**, followed by the command and its expanded arguments or associated word list, to the standard error.

−B The shell performs brace expansion (see **Brace Expansion** above). This is on by default.

−C If set, **bash** does not overwrite an existing file with the **>**, **>&**, and **<>** redirection operators. Using the redirection operator **>|** instead of **>** will override this and force the creation of an output file.

−E If set, any trap on **ERR** is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The **ERR** trap is normally not inherited in such cases.

−H Enable **!** style history substitution. This option is on by default when the shell is interactive.

−P If set, the shell does not resolve symbolic links when executing commands such as **cd** that change the current working directory. It uses the physical directory structure instead. By default, **bash** follows the logical chain of directories when performing commands which change the current directory.

−T If set, any traps on **DEBUG** and **RETURN** are inherited by shell functions, command substitutions, and commands executed in a subshell environment. The **DEBUG** and **RETURN** traps are normally not inherited in such cases.

−− If no arguments follow this option, unset the positional parameters. Otherwise, set the positional parameters to the *arg*s, even if some of them begin with a **−**.

− Signal the end of options, and assign all remaining *arg*s to the positional parameters. The **−x** and **−v** options are turned off. If there are no *arg*s, the positional parameters remain unchanged.

The options are off by default unless otherwise noted. Using + rather than − causes these options to be turned off. The options can also be specified as arguments to an invocation of the shell. The current set of options may be found in **$−**. The return status is always zero unless an invalid option is encountered.

shift [*n*]

Rename positional parameters from *n*+1 ... to **$1 ....** Parameters represented by the numbers **$#** down to **$#**−*n*+1 are unset. *n* must be a non-negative number less than or equal to **$#**. If *n* is 0, no parameters are changed. If *n* is not given, it is assumed to be 1. If *n* is greater than **$#**, the positional parameters are not changed. The return status is greater than zero if *n* is greater than **$#** or less than zero; otherwise 0.

shopt [**−pqsu**] [**−o**] [*optname* ...]

Toggle the values of settings controlling optional shell behavior. The settings can be either those listed below, or, if the **−o** option is used, those available with the **−o** option to the **set** builtin command.

With no options, or with the **−p** option, display a list of all settable options, with an indication of whether or not each is set; if any *optnames* are supplied, the output is restricted to those options. The **−p** option displays output in a form that may be reused as input.

Other options have the following meanings:

−s Enable (set) each *optname*.

−u Disable (unset) each *optname*.

−q Suppresses normal output (quiet mode); the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are supplied with **−q**, the return status is zero if all *optnames* are enabled; non-zero otherwise.

−o Restricts the values of *optname* to be those defined for the **−o** option to the **set** builtin.

If either **−s** or **−u** is used with no *optname* arguments, **shopt** shows only those options which are set or unset, respectively. Unless otherwise noted, the **shopt** options are disabled (unset) by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a valid shell option.

The list of **shopt** options is:

array_expand_once

If set, the shell suppresses multiple evaluation of associative and indexed array subscripts during arithmetic expression evaluation, while executing builtins that can perform variable assignments, and while executing builtins that perform array dereferencing.

assoc_expand_once

Deprecated; a synonym for **array_expand_once**.

autocd If set, a command name that is the name of a directory is executed as if it were the argument to the **cd** command. This option is only used by interactive shells.

bash_source_fullpath

If set, filenames added to the **BASH_SOURCE** array variable are converted to full pathnames (see **Shell Variables** above).

cdable_vars

If set, an argument to the **cd** builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.

cdspell If set, the **cd** command attempts to correct minor errors in the spelling of a directory component. Minor errors include transposed characters, a missing character, and one extra character. If **cd** corrects the directory name, it prints the corrected filename, and the command proceeds. This option is only used by interactive shells.

checkhash

If set, **bash** checks that a command found in the hash table exists before trying to execute it. If a hashed command no longer exists, **bash** performs a normal path search.

checkjobs

If set, **bash** lists the status of any stopped and running jobs before exiting an interactive shell. If any jobs are running, **bash** defers the exit until a second exit is attempted without an intervening command (see **JOB CONTROL** above). The shell always postpones exiting if any jobs are stopped.

checkwinsize
If set, **bash** checks the window size after each external (non-builtin) command and, if necessary, updates the values of **LINES** and **COLUMNS**, using the file descriptor associated with the standard error if it is a terminal. This option is enabled by default.

cmdhist If set, **bash** attempts to save all lines of a multiple-line command in the same history entry. This allows easy re-editing of multi-line commands. This option is enabled by default, but only has an effect if command history is enabled, as described above under **HISTORY**.

compat31
compat32
compat40
compat41
compat42
compat43
compat44
These control aspects of the shell's compatibility mode (see **SHELL COMPATIBILITY MODE** below).

complete_fullquote
If set, **bash** quotes all shell metacharacters in filenames and directory names when performing completion. If not set, **bash** removes metacharacters such as the dollar sign from the set of characters that will be quoted in completed filenames when these metacharacters appear in shell variable references in words to be completed. This means that dollar signs in variable names that expand to directories will not be quoted; however, any dollar signs appearing in filenames will not be quoted, either. This is active only when bash is using backslashes to quote completed filenames. This variable is set by default, which is the default bash behavior in versions through 4.2.

direxpand
If set, **bash** replaces directory names with the results of word expansion when performing filename completion. This changes the contents of the **readline** editing buffer. If not set, **bash** attempts to preserve what the user typed.

dirspell If set, **bash** attempts spelling correction on directory names during word completion if the directory name initially supplied does not exist.

dotglob If set, **bash** includes filenames beginning with a "." in the results of pathname expansion. The filenames . and .. must always be matched explicitly, even if **dotglob** is set.

execfail If set, a non-interactive shell will not exit if it cannot execute the file specified as an argument to the **exec** builtin. An interactive shell does not exit if **exec** fails.

expand_aliases
If set, aliases are expanded as described above under **ALIASES**. This option is enabled by default for interactive shells.

extdebug
If set at shell invocation, or in a shell startup file, arrange to execute the debugger profile before the shell starts, identical to the **−−debugger** option. If set after invocation, behavior intended for use by debuggers is enabled:

1. The **−F** option to the **declare** builtin displays the source file name and line number corresponding to each function name supplied as an argument.
2. If the command run by the **DEBUG** trap returns a non-zero value, the next command is skipped and not executed.
3. If the command run by the **DEBUG** trap returns a value of 2, and the shell is executing in a subroutine (a shell function or a shell script executed by the **.** or **source** builtins), the shell simulates a call to **return**.
4. **BASH_ARGC** and **BASH_ARGV** are updated as described in their descriptions above).

5. Function tracing is enabled: command substitution, shell functions, and subshells invoked with **(** *command* **)** inherit the **DEBUG** and **RETURN** traps.

6. Error tracing is enabled: command substitution, shell functions, and subshells invoked with **(** *command* **)** inherit the **ERR** trap.

extglob If set, enable the extended pattern matching features described above under **Pathname Expansion**.

extquote
If set, **$**'*string*' and **$**"*string*" quoting is performed within **${***parameter***}** expansions enclosed in double quotes. This option is enabled by default.

failglob If set, patterns which fail to match filenames during pathname expansion result in an expansion error.

force_fignore
If set, the suffixes specified by the **FIGNORE** shell variable cause words to be ignored when performing word completion even if the ignored words are the only possible completions. See **Shell Variables** above for a description of **FIGNORE**. This option is enabled by default.

globasciiranges
If set, range expressions used in pattern matching bracket expressions (see **Pattern Matching** above) behave as if in the traditional C locale when performing comparisons. That is, pattern matching does not take the current locale's collating sequence into account, so **b** will not collate between **A** and **B**, and upper-case and lower-case ASCII characters will collate together.

globskipdots
If set, pathname expansion will never match the filenames . and .. , even if the pattern begins with a ".". This option is enabled by default.

globstar If set, the pattern ****** used in a pathname expansion context will match all files and zero or more directories and subdirectories. If the pattern is followed by a **/**, only directories and subdirectories match.

gnu_errfmt
If set, shell error messages are written in the standard GNU error message format.

histappend
If set, the history list is appended to the file named by the value of the **HISTFILE** variable when the shell exits, rather than overwriting the file.

histreedit
If set, and **readline** is being used, the user is given the opportunity to re-edit a failed history substitution.

histverify
If set, and **readline** is being used, the results of history substitution are not immediately passed to the shell parser. Instead, the resulting line is loaded into the **readline** editing buffer, allowing further modification.

hostcomplete
If set, and **readline** is being used, **bash** will attempt to perform hostname completion when a word containing a **@** is being completed (see **Completing** under **READLINE** above). This is enabled by default.

huponexit
If set, **bash** will send **SIGHUP** to all jobs when an interactive login shell exits.

inherit_errexit
If set, command substitution inherits the value of the **errexit** option, instead of unsetting it in the subshell environment. This option is enabled when posix mode is enabled.

interactive_comments
In an interactive shell, a word beginning with **#** causes that word and all remaining characters on that line to be ignored, as in a non-interactive shell (see **COMMENTS** above). This option is enabled by default.

lastpipe If set, and job control is not active, the shell runs the last command of a pipeline not executed in the background in the current shell environment.

lithist If set, and the **cmdhist** option is enabled, multi-line commands are saved to the history with embedded newlines rather than using semicolon separators where possible.

localvar_inherit
If set, local variables inherit the value and attributes of a variable of the same name that exists at a previous scope before any new value is assigned. The nameref attribute is not inherited.

localvar_unset
If set, calling **unset** on local variables in previous function scopes marks them so subsequent lookups find them unset until that function returns. This is identical to the behavior of unsetting local variables at the current function scope.

login_shell
The shell sets this option if it is started as a login shell (see **INVOCATION** above). The value may not be changed.

mailwarn
If set, and a file that **bash** is checking for mail has been accessed since the last time it was checked, **bash** displays the message "The mail in *mailfile* has been read".

no_empty_cmd_completion
If set, and **readline** is being used, **bash** does not search **PATH** for possible completions when completion is attempted on an empty line.

nocaseglob
If set, **bash** matches filenames in a case−insensitive fashion when performing pathname expansion (see **Pathname Expansion** above).

nocasematch
If set, **bash** matches patterns in a case−insensitive fashion when performing matching while executing **case** or **[[** conditional commands, when performing pattern substitution word expansions, or when filtering possible completions as part of programmable completion.

noexpand_translation
If set, **bash** encloses the translated results of **$"**...**"** quoting in single quotes instead of double quotes. If the string is not translated, this has no effect.

nullglob
If set, pathname expansion patterns which match no files (see **Pathname Expansion** above) expand to nothing and are removed, rather than expanding to themselves.

patsub_replacement
If set, **bash** expands occurrences of **&** in the replacement string of pattern substitution to the text matched by the pattern, as described under **Parameter Expansion** above. This option is enabled by default.

progcomp
If set, enable the programmable completion facilities (see **Programmable Completion** above). This option is enabled by default.

progcomp_alias
If set, and programmable completion is enabled, **bash** treats a command name that doesn't have any completions as a possible alias and attempts alias expansion. If it has an alias, **bash** attempts programmable completion using the command word resulting from the expanded alias.

promptvars
If set, prompt strings undergo parameter expansion, command substitution, arithmetic expansion, and quote removal after being expanded as described in **PROMPTING** above. This option is enabled by default.

restricted_shell
The shell sets this option if it is started in restricted mode (see **RESTRICTED SHELL** below). The value may not be changed. This is not reset when the startup files are

executed, allowing the startup files to discover whether or not a shell is restricted.

shift_verbose
If set, the **shift** builtin prints an error message when the shift count exceeds the number of positional parameters.

sourcepath
If set, the **.** (**source**) builtin uses the value of **PATH** to find the directory containing the file supplied as an argument when the **−p** option is not supplied. This option is enabled by default.

varredir_close
If set, the shell automatically closes file descriptors assigned using the *{varname}* redirection syntax (see **REDIRECTION** above) instead of leaving them open when the command completes.

xpg_echo
If set, the **echo** builtin expands backslash-escape sequences by default. If the **posix** shell option is also enabled, **echo** does not interpret any options.

suspend [**−f**]
Suspend the execution of this shell until it receives a **SIGCONT** signal. A login shell, or a shell without job control enabled, cannot be suspended; the **−f** option will override this and force the suspension. The return status is 0 unless the shell is a login shell or job control is not enabled and **−f** is not supplied.

test *expr*
[*expr* **]** Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression *expr*. Each operator and operand must be a separate argument. Expressions are composed of the primaries described above under **CONDITIONAL EXPRESSIONS**. **test** does not accept any options, nor does it accept and ignore an argument of **−−** as signifying the end of options.

Expressions may be combined using the following operators, listed in decreasing order of precedence. The evaluation depends on the number of arguments; see below. **test** uses operator precedence when there are five or more arguments.

! *expr* True if *expr* is false.
(*expr* **)** Returns the value of *expr*. This may be used to override normal operator precedence.
expr1 **−a** *expr2*
True if both *expr1* and *expr2* are true.
expr1 **−o** *expr2*
True if either *expr1* or *expr2* is true.

test and **[** evaluate conditional expressions using a set of rules based on the number of arguments.

0 arguments
The expression is false.
1 argument
The expression is true if and only if the argument is not null.
2 arguments
If the first argument is **!**, the expression is true if and only if the second argument is null. If the first argument is one of the unary conditional operators listed above under **CONDITIONAL EXPRESSIONS**, the expression is true if the unary test is true. If the first argument is not a valid unary conditional operator, the expression is false.
3 arguments
The following conditions are applied in the order listed. If the second argument is one of the binary conditional operators listed above under **CONDITIONAL EXPRESSIONS**, the result of the expression is the result of the binary test using the first and third arguments as operands. The **−a** and **−o** operators are considered binary operators when there are three arguments. If the first argument is **!**, the value is the negation of the two-argument test using the second and third arguments. If the first argument is exactly (and the third argument is exactly), the result is the one-argument test of the second argument.

Otherwise, the expression is false.

4 arguments

The following conditions are applied in the order listed. If the first argument is **!**, the result is the negation of the three-argument expression composed of the remaining arguments. If the first argument is exactly **(** and the fourth argument is exactly **)**, the result is the two-argument test of the second and third arguments. Otherwise, the expression is parsed and evaluated according to precedence using the rules listed above.

5 or more arguments

The expression is parsed and evaluated according to precedence using the rules listed above.

When the shell is in posix mode, or if the expression is part of the **[[** command, the **<** and **>** operators sort using the current locale. If the shell is not in posix mode, the **test** and **[** commands sort lexicographically using ASCII ordering.

The historical operator-precedence parsing with 4 or more arguments can lead to ambiguities when it encounters strings that look like primaries. The POSIX standard has deprecated the **−a** and **−o** primaries and enclosing expressions within parentheses. Scripts should no longer use them. It's much more reliable to restrict test invocations to a single primary, and to replace uses of **−a** and **−o** with the shell's **&&** and **||** list operators.

times Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.

trap [**−lpP**] [[*action*] *sigspec* . . .]

The *action* is a command that is read and executed when the shell receives any of the signals *sigspec*. If *action* is absent (and there is a single *sigspec*) or **−**, each specified *sigspec* is reset to the value it had when the shell was started. If *action* is the null string the signal specified by each *sigspec* is ignored by the shell and by the commands it invokes.

If no arguments are supplied, **trap** displays the actions associated with each trapped signal as a set of **trap** commands that can be reused as shell input to restore the current signal dispositions. If **−p** is given, and *action* is not present, then **trap** displays the actions associated with each *sigspec* or, if none are supplied, for all trapped signals, as a set of **trap** commands that can be reused as shell input to restore the current signal dispositions. The **−P** option behaves similarly, but displays only the actions associated with each *sigspec* argument. **−P** requires at least one *sigspec* argument. The **−P** or **−p** options may be used in a subshell environment (e.g., command substitution) and, as long as they are used before **trap** is used to change a signal's handling, will display the state of its parent's traps.

The **−l** option prints a list of signal names and their corresponding numbers. Each *sigspec* is either a signal name defined in <*signal.h*>, or a signal number. Signal names are case insensitive and the **SIG** prefix is optional. If **−l** is supplied with no *sigspec* arguments, it prints a list of valid signal names.

If a *sigspec* is **EXIT** (0), *action* is executed on exit from the shell. If a *sigspec* is **DEBUG**, *action* is executed before every *simple command*, *for* command, *case* command, *select* command, ((arithmetic command, [[conditional command, arithmetic *for* command, and before the first command executes in a shell function (see **SHELL GRAMMAR** above). Refer to the description of the **extdebug** shell option (see **shopt** above) for details of its effect on the **DEBUG** trap. If a *sigspec* is **RETURN**, *action* is executed each time a shell function or a script executed with the **.** or **source** builtins finishes executing.

If a *sigspec* is **ERR**, *action* is executed whenever a pipeline (which may consist of a single simple command), a list, or a compound command returns a non−zero exit status, subject to the following conditions. The **ERR** trap is not executed if the failed command is part of the command list immediately following a **while** or **until** reserved word, part of the test in an *if* statement, part of a command executed in a **&&** or **||** list except the command following the final **&&** or **||**, any command in a pipeline but the last (subject to the state of the **pipefail** shell option), or if the command's return value is being inverted using **!**. These are the same conditions obeyed by the **errexit** (**−e**) option.

When the shell is not interactive, signals ignored upon entry to the shell cannot be trapped or reset. Interactive shells permit trapping signals ignored on entry. Trapped signals that are not being ignored are reset to their original values in a subshell or subshell environment when one is created. The return status is false if any *sigspec* is invalid; otherwise **trap** returns true.

true Does nothing, returns a 0 status.

type [**−aftpP**] *name* [*name* . . .]

Indicate how each *name* would be interpreted if used as a command name.

If the **−t** option is used, **type** prints a string which is one of *alias*, *keyword*, *function*, *builtin*, or *file* if *name* is an alias, shell reserved word, function, builtin, or executable file, respectively. If the *name* is not found, **type** prints nothing and returns a non-zero exit status.

If the **−p** option is used, **type** either returns the pathname of the executable file that would be found by searching **$PATH** for *name* or nothing if "type −t name" would not return *file*. The **−P** option forces a **PATH** search for each *name*, even if "type −t name" would not return *file*. If *name* is present in the table of hashed commands, **−p** and **−P** print the hashed value, which is not necessarily the file that appears first in **PATH**.

If the **−a** option is used, **type** prints all of the places that contain a command named *name*. This includes aliases, reserved words, functions, and builtins, but the path search options (**−p** and **−P**) can be supplied to restrict the output to executable files. **type** does not consult the table of hashed commands when using **−a** with **−p**, and only performs a **PATH** search for *name*.

The **−f** option suppresses shell function lookup, as with the **command** builtin. **type** returns true if all of the arguments are found, false if any are not found.

ulimit [**−HS**] **−a**
ulimit [**−HS**] [**−bcdefiklmnpqrstuvxPRT** [*limit*]]

Provides control over the resources available to the shell and to processes it starts, on systems that allow such control.

The **−H** and **−S** options specify whether the hard or soft limit is set for the given resource. A hard limit cannot be increased by a non-root user once it is set; a soft limit may be increased up to the value of the hard limit. If neither **−H** nor **−S** is specified, **ulimit** sets both the soft and hard limits.

The value of *limit* can be a number in the unit specified for the resource or one of the special values **hard**, **soft**, or **unlimited**, which stand for the current hard limit, the current soft limit, and no limit, respectively. If *limit* is omitted, **ulimit** prints the current value of the soft limit of the resource, unless the **−H** option is given. When more than one resource is specified, the limit name and unit, if appropriate, are printed before the value. Other options are interpreted as follows:

−a Report all current limits; no limits are set.
−b The maximum socket buffer size.
−c The maximum size of core files created.
−d The maximum size of a process's data segment.
−e The maximum scheduling priority ("nice").
−f The maximum size of files written by the shell and its children.
−i The maximum number of pending signals.
−k The maximum number of kqueues that may be allocated.
−l The maximum size that may be locked into memory.
−m The maximum resident set size (many systems do not honor this limit).
−n The maximum number of open file descriptors (most systems do not allow this value to be set).
−p The pipe size in 512-byte blocks (this may not be set).

- **−q** The maximum number of bytes in POSIX message queues.
- **−r** The maximum real-time scheduling priority.
- **−s** The maximum stack size.
- **−t** The maximum amount of cpu time in seconds.
- **−u** The maximum number of processes available to a single user.
- **−v** The maximum amount of virtual memory available to the shell and, on some systems, to its children.
- **−x** The maximum number of file locks.
- **−P** The maximum number of pseudoterminals.
- **−R** The maximum time a real-time process can run before blocking, in microseconds.
- **−T** The maximum number of threads.

If *limit* is supplied, and the **−a** option is not used, *limit* is the new value of the specified resource. If no option is supplied, then **−f** is assumed.

Values are in 1024-byte increments, except for **−t**, which is in seconds; **−R**, which is in microseconds; **−p**, which is in units of 512-byte blocks; **−P**, **−T**, **−b**, **−k**, **−n**, and **−u**, which are unscaled values; and, when in posix mode, **−c** and **−f**, which are in 512-byte increments. The return status is 0 unless an invalid option or argument is supplied, or an error occurs while setting a new limit.

umask [**−p**] [**−S**] [*mode*]

Set the user file-creation mask to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by *chmod*(1). If *mode* is omitted, **umask** prints the current value of the mask. The **−S** option without a *mode* argument prints the mask in a symbolic format; the default output is an octal number. If the **−p** option is supplied, and *mode* is omitted, the output is in a form that may be reused as input. The return status is zero if the mode was successfully changed or if no *mode* argument was supplied, and non-zero otherwise.

unalias [**−a**] [*name* ...]

Remove each *name* from the list of defined aliases. If **−a** is supplied, remove all alias definitions. The return value is true unless a supplied *name* is not a defined alias.

unset [**−fv**] [**−n**] [*name* ...]

For each *name*, remove the corresponding variable or function. If the **−v** option is given, each *name* refers to a shell variable, and that variable is removed. If **−f** is specified, each *name* refers to a shell function, and the function definition is removed. If the **−n** option is supplied, and *name* is a variable with the *nameref* attribute, *name* will be unset rather than the variable it references. **−n** has no effect if the **−f** option is supplied. Read-only variables and functions may not be unset. When variables or functions are removed, they are also removed from the environment passed to subsequent commands. If no options are supplied, each *name* refers to a variable; if there is no variable by that name, a function with that name, if any, is unset. Some shell variables may not be unset. If any of **BASH_ALIASES**, **BASH_ARGV0**, **BASH_CMDS**, **BASH_COMMAND**, **BASH_SUBSHELL**, **BASHPID**, **COMP_WORDBREAKS**, **DIRSTACK**, **EPOCHREALTIME**, **EPOCHSECONDS**, **FUNCNAME**, **GROUPS**, **HISTCMD**, **LINENO**, **RANDOM**, **SECONDS**, or **SRANDOM** are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a *name* is readonly or may not be unset.

wait [**−fn**] [**−p** *varname*] [*id* ...]

Wait for each specified child process *id* and return the termination status of the last *id*. Each *id* may be a process ID *pid* or a job specification *jobspec*; if a jobspec is supplied, **wait** waits for all processes in the job.

If no options or *id*s are supplied, **wait** waits for all running background jobs and the last-executed process substitution, if its process id is the same as **$!**, and the return status is zero.

If the **−n** option is supplied, **wait** waits for any one of the given *id*s or, if no *id*s are supplied, any job or process substitution, to complete and returns its exit status. If none of the supplied *id*s is a child of the shell, or if no *id*s are supplied and the shell has no unwaited-for children, the exit status is 127.

If the **−p** option is supplied, **wait** assigns the process or job identifier of the job for which the exit status is returned to the variable *varname* named by the option argument. The variable, which cannot be readonly, will be unset initially, before any assignment. This is useful only when used with the **−n** option.

Supplying the **−f** option, when job control is enabled, forces **wait** to wait for each *id* to terminate before returning its status, instead of returning when it changes status.

If none of the *id*s specify one of the shell's active child processes, the return status is 127. If **wait** is interrupted by a signal, any *varname* will remain unset, and the return status will be greater than 128, as described under **SIGNALS** above. Otherwise, the return status is the exit status of the last *id*.

SHELL COMPATIBILITY MODE

Bash-4.0 introduced the concept of a *shell compatibility level*, specified as a set of options to the shopt builtin (**compat31**, **compat32**, **compat40**, **compat41**, and so on). There is only one current compatibility level — each option is mutually exclusive. The compatibility level is intended to allow users to select behavior from previous versions that is incompatible with newer versions while they migrate scripts to use current features and behavior. It's intended to be a temporary solution.

This section does not mention behavior that is standard for a particular version (e.g., setting **compat32** means that quoting the right hand side of the regexp matching operator quotes special regexp characters in the word, which is default behavior in bash-3.2 and subsequent versions).

If a user enables, say, **compat32**, it may affect the behavior of other compatibility levels up to and including the current compatibility level. The idea is that each compatibility level controls behavior that changed in that version of **bash**, but that behavior may have been present in earlier versions. For instance, the change to use locale-based comparisons with the **[[** command came in bash-4.1, and earlier versions used ASCII-based comparisons, so enabling **compat32** will enable ASCII-based comparisons as well. That granularity may not be sufficient for all uses, and as a result users should employ compatibility levels carefully. Read the documentation for a particular feature to find out the current behavior.

Bash-4.3 introduced a new shell variable: **BASH_COMPAT**. The value assigned to this variable (a decimal version number like 4.2, or an integer corresponding to the **compat***NN* option, like 42) determines the compatibility level.

Starting with bash-4.4, **bash** began deprecating older compatibility levels. Eventually, the options will be removed in favor of **BASH_COMPAT**.

Bash-5.0 was the final version for which there was an individual shopt option for the previous version. **BASH_COMPAT** is the only mechanism to control the compatibility level in versions newer than bash-5.0.

The following table describes the behavior changes controlled by each compatibility level setting. The **compat***NN* tag is used as shorthand for setting the compatibility level to *NN* using one of the following mechanisms. For versions prior to bash-5.0, the compatibility level may be set using the corresponding **compat***NN* shopt option. For bash-4.3 and later versions, the **BASH_COMPAT** variable is preferred, and it is required for bash-5.1 and later versions.

compat31

- Quoting the rhs of the **[[** command's regexp matching operator (=~) has no special effect.

compat32

- The **<** and **>** operators to the **[[** command do not consider the current locale when comparing strings; they use ASCII ordering.

compat40

- The **<** and **>** operators to the **[[** command do not consider the current locale when comparing strings; they use ASCII ordering. **Bash** versions prior to bash-4.1 use ASCII collation and *strcmp*(3); bash-4.1 and later use the current locale's collation sequence and *strcoll*(3).

compat41

- In posix mode, **time** may be followed by options and still be recognized as a reserved word (this is POSIX interpretation 267).
- In *posix* mode, the parser requires that an even number of single quotes occur in the *word* portion of a double-quoted parameter expansion and treats them specially, so that characters within the single quotes are considered quoted (this is POSIX interpretation 221).

compat42

- The replacement string in double-quoted pattern substitution does not undergo quote removal, as it does in versions after bash-4.2.
- In posix mode, single quotes are considered special when expanding the *word* portion of a double-quoted parameter expansion and can be used to quote a closing brace or other special character (this is part of POSIX interpretation 221); in later versions, single quotes are not special within double-quoted word expansions.

compat43

- Word expansion errors are considered non-fatal errors that cause the current command to fail, even in posix mode (the default behavior is to make them fatal errors that cause the shell to exit).
- When executing a shell function, the loop state (while/until/etc.) is not reset, so **break** or **continue** in that function will break or continue loops in the calling context. Bash-4.4 and later reset the loop state to prevent this.

compat44

- The shell sets up the values used by **BASH_ARGV** and **BASH_ARGC** so they can expand to the shell's positional parameters even if extended debugging mode is not enabled.
- A subshell inherits loops from its parent context, so **break** or **continue** will cause the subshell to exit. Bash-5.0 and later reset the loop state to prevent the exit
- Variable assignments preceding builtins like **export** and **readonly** that set attributes continue to affect variables with the same name in the calling environment even if the shell is not in posix mode.

compat50

- Bash-5.1 changed the way **$RANDOM** is generated to introduce slightly more randomness. If the shell compatibility level is set to 50 or lower, it reverts to the method from bash-5.0 and previous versions, so seeding the random number generator by assigning a value to **RANDOM** will produce the same sequence as in bash-5.0.
- If the command hash table is empty, bash versions prior to bash-5.1 printed an informational message to that effect, even when producing output that can be reused as input. Bash-5.1 suppresses that message when the **–l** option is supplied.

compat51

- The **unset** builtin treats attempts to unset array subscripts @ and * differently depending on whether the array is indexed or associative, and differently than in previous versions.
- Arithmetic commands (**((...))**) and the expressions in an arithmetic for statement can be expanded more than once.
- Expressions used as arguments to arithmetic operators in the **[[** conditional command can be expanded more than once.
- The expressions in substring parameter brace expansion can be expanded more than once.
- The expressions in the **$((...))** word expansion can be expanded more than once.
- Arithmetic expressions used as indexed array subscripts can be expanded more than once.
- **test –v**, when given an argument of **A[@]**, where **A** is an existing associative array, will return true if the array has any set elements. Bash-5.2 will look for and report on a key named @.
- The ${*parameter***[:]=***value*} word expansion will return *value*, before any variable-specific transformations have been performed (e.g., converting to lowercase). Bash-5.2 will return the final value assigned to the variable.

- Parsing command substitutions will behave as if extended globbing (see the description of the **shopt** builtin above) is enabled, so that parsing a command substitution containing an extglob pattern (say, as part of a shell function) will not fail. This assumes the intent is to enable extglob before the command is executed and word expansions are performed. It will fail at word expansion time if extglob hasn't been enabled by the time the command is executed.

compat52

- The **test** builtin uses its historical algorithm to parse parenthesized subexpressions when given five or more arguments.
- If the **−p** or **−P** option is supplied to the **bind** builtin, **bind** treats any arguments remaining after option processing as bindable command names, and displays any key sequences bound to those commands, instead of treating the arguments as key sequences to bind.

RESTRICTED SHELL

If **bash** is started with the name **rbash**, or the **−r** option is supplied at invocation, the shell becomes *restricted*. A restricted shell is used to set up an environment more controlled than the standard shell. It behaves identically to **bash** with the exception that the following are disallowed or not performed:

- Changing directories with **cd**.
- Setting or unsetting the values of **SHELL**, **PATH**, **HISTFILE**, **ENV**, or **BASH_ENV**.
- Specifying command names containing **/**.
- Specifying a filename containing a **/** as an argument to the **.** builtin command.
- Using the **−p** option to the **.** builtin command to specify a search path.
- Specifying a filename containing a slash as an argument to the **history** builtin command.
- Specifying a filename containing a slash as an argument to the **−p** option to the **hash** builtin command.
- Importing function definitions from the shell environment at startup.
- Parsing the values of **BASHOPTS** and **SHELLOPTS** from the shell environment at startup.
- Redirecting output using the >, >|, <>, >&, &>, and >> redirection operators.
- Using the **exec** builtin command to replace the shell with another command.
- Adding or deleting builtin commands with the **−f** and **−d** options to the **enable** builtin command.
- Using the **enable** builtin command to enable disabled shell builtins.
- Specifying the **−p** option to the **command** builtin command.
- Turning off restricted mode with **set +r** or **shopt −u restricted_shell**.

These restrictions are enforced after any startup files are read.

When a command that is found to be a shell script is executed (see **COMMAND EXECUTION** above), **rbash** turns off any restrictions in the shell spawned to execute the script.

SEE ALSO

Bash Reference Manual, Brian Fox and Chet Ramey
The Gnu Readline Library, Brian Fox and Chet Ramey
The Gnu History Library, Brian Fox and Chet Ramey
Portable Operating System Interface (POSIX) Part 2: Shell and Utilities, IEEE —
    http://pubs.opengroup.org/onlinepubs/9799919799/
http://tiswww.case.edu/~chet/bash/POSIX — a description of posix mode
sh(1), *ksh*(1), *csh*(1)
emacs(1), *vi*(1)

readline(3)

FILES

/bin/bash
: The **bash** executable

/etc/profile
: The systemwide initialization file, executed for login shells

~/.bash_profile
: The personal initialization file, executed for login shells

~/.bashrc
: The individual per-interactive-shell startup file

~/.bash_logout
: The individual login shell cleanup file, executed when a login shell exits

~/.bash_history
: The default value of **HISTFILE**, the file in which bash saves the command history

~/.inputrc
: Individual *readline* initialization file

AUTHORS

Brian Fox, Free Software Foundation
bfox@gnu.org

Chet Ramey, Case Western Reserve University
chet.ramey@case.edu

BUG REPORTS

If you find a bug in **bash**, you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of **bash**. The latest version is always available from *ftp://ftp.gnu.org/pub/gnu/bash/* and *http://git.savannah.gnu.org/cgit/bash.git/snapshot/bash-master.tar.gz*.

Once you have determined that a bug actually exists, use the *bashbug* command to submit a bug report. If you have a fix, you are encouraged to mail that as well! You may send suggestions and "philosophical" bug reports to *bug-bash@gnu.org* or post them to the Usenet newsgroup **gnu.bash.bug**.

ALL bug reports should include:

The version number of **bash**
The hardware and operating system
The compiler used to compile
A description of the bug behavior
A short script or "recipe" which exercises the bug

bashbug inserts the first three items automatically into the template it provides for filing a bug report.

Comments and bug reports concerning this manual page should be directed to *chet.ramey@case.edu*.

BUGS

It's too big and too slow.

There are some subtle differences between **bash** and traditional versions of **sh**, mostly because of the POSIX specification.

Aliases are confusing in some uses.

Shell builtin commands and functions are not stoppable/restartable.

Compound commands and command lists of the form "a ; b ; c" are not handled gracefully when combined with process suspension. When a process is stopped, the shell immediately executes the next command in the list or breaks out of any existing loops. It suffices to enclose the command in parentheses to force it into a subshell, which may be stopped as a unit, or to start the command in the background and immediately bring it into the foreground.

Array variables may not (yet) be exported.