NAME

bash − GNU Bourne−Again SHell

SYNOPSIS

bash [options] [file]

COPYRIGHT

DESCRIPTION

Bash is an **sh**−compatible command language interpreter that executes commands read from the standard input or from a file. **Bash** also incorporates useful features from the *Korn* and *C* shells (**ksh** and **csh**).

Bash is ultimately intended to be a conformant implementation of the IEEE Posix Shell and Tools specification (IEEE Working Group 1003.2).

OPTIONS

In addition to the single−character shell options documented in the description of the **set** builtin command, **bash** interprets the following flags when it is invoked:

−c *string* If the **−c** flag is present, then commands are read from *string*. If there are arguments after the *string*, they are assigned to the positional parameters, starting with **$0**.

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

−s If the **−s** flag is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell.

− A single **−** signals the end of options and disables further option processing. Any arguments after the **−** are treated as filenames and arguments. An argument of **−−** is equivalent to an argument of **−**.

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.

−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**.

−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** normally reads these files when it is invoked as a login shell (see **INVOCATION** below).

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

−version Show the version number of this instance of **bash** when starting.

−quiet Do not be verbose when starting up (do not show the shell version or any other information). This is the default.

−login Make **bash** act as if it had been invoked as a login shell.

−nobraceexpansion
Do not perform curly brace expansion (see **Brace Expansion** below).

−nolineediting
Do not use the GNU *readline* library to read command lines if interactive.

−posix Change the behavior of bash where the default operation differs from the Posix 1003.2 standard to match the standard

ARGUMENTS

If arguments remain after option processing, and neither the **−c** nor the **−s** option has been supplied, the first argument is assumed to be the name of a file containing shell commands. If **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.

DEFINITIONS

blank A space or tab.

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**

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 simple command (see **SHELL GRAMMAR** below) or the third word of a **case** or **for** command:

> **! case do done elif else esac fi for function if in select then until while { }**

SHELL GRAMMAR

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. 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 the character **|**. The format for a pipeline is:

> [!] *command* [**|** *command2* ...]

The standard output of *command* is connected to the standard input of *command2*. This connection is performed before any redirections specified by the command (see **REDIRECTION** below).

If the reserved word **!** precedes a pipeline, the exit status of that pipeline is the logical NOT of the exit status of the last command. Otherwise, the status of the pipeline is the exit status of the last command. The shell waits for all commands in the pipeline to terminate before returning a value.

Each command in a pipeline is executed as a separate process (i.e., in a subshell).

Lists

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

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

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. 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.

The control operators **&&** and **||** denote AND lists and OR lists, respectively. An AND list has the form

> *command* **&&** *command2*

command2 is executed if, and only if, *command* returns an exit status of zero.

An OR list has the form

> *command* **||** *command2*

command2 is executed if and only if *command* 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:

(*list*) *list* is executed in a subshell. 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 simply executed in the current shell environment. This is known as a *group command*. The return status is the exit status of *list*.

for *name* [**in** *word*;] **do** *list* ; **done**
The list of words following **in** is expanded, generating a list of items. 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).

select *name* [**in** *word*;] **do** *list* ; **done**
The list of words following **in** is expanded, generating a list of items. The set of expanded words is printed on the standard error, each preceded by a number. If the **in** *word* is omitted, the positional parameters are printed (see **PARAMETERS** below). The **PS3** prompt is then displayed and a line read from the standard input. If the line consists of the number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If EOF is read, the command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable **REPLY**. The *list* is executed after each selection until a **break** or **return** 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, using the same matching rules as for pathname expansion (see **Pathname Expansion** below). When a match is found, the corresponding *list* is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no patterns are matches. Otherwise, it is the exit status of the last command executed in *list*.

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

[**function**] *name* () { *list*; }
This defines a function named *name*. The *body* of the function is the *list* of commands between { and }. This list is executed whenever *name* is specified as the name of a simple command. 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 **-o interactive−comments** option to the **set** builtin is enabled, a word beginning with **#** causes that word and all remaining characters on that line to be ignored. An interactive shell without the **-o interactive−comments** option enabled does not allow comments.

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 they are to represent themselves. There are three quoting mechanisms: the *escape character*, single quotes, and double quotes.

A non-quoted backslash (\) is the *escape character*. It preserves the literal value of the next character that follows, with the exception of <newline>. If a \<newline> pair appears, and the backslash is not quoted, the \<newline> is treated as a line continuation (that is, it is 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 **$**, **‘**, and ****. The characters **$** 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>**. A double quote may be quoted within double quotes by preceding it with a backslash.

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

PARAMETERS

A *parameter* is an entity that stores values, somewhat like a variable in a conventional programming language. It can be a *name*, a number, or one of the special characters listed below under **Special Parameters**. For the shell's purposes, a *variable* is a parameter denoted by a *name*.

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* may be assigned to by 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. If the variable has its **−i** attribute set (see **declare** below in **SHELL BUILTIN COMMANDS**) then *value* is subject to arithmetic expansion even if the $[...] syntax does not appear. Word splitting is not performed, with the exception of **"$@"** as explained below under **Special Parameters**. Pathname expansion is not performed.

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).

Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

- ***** Expands to the positional parameters, starting from one. 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** special variable. That is, “**$***” is equivalent to “**$1***c***$2***c***...**”, where *c* is the first character of the value of the **IFS** variable. If **IFS** is null or unset, the parameters are separated by spaces.
- **@** Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands as a separate word. That is, “ **$@**” is equivalent to “**$1**” “**$2**” ... When there are no positional parameters, “**$@**” and **$@** expand to nothing (i.e., they are

removed).

Expands to the number of positional parameters in decimal.

? Expands to the status of the most recently executed foreground pipeline.

– 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** flag).

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

! Expands to the process ID of the most recently executed background (asynchronous) command.

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 pathname used to invoke **bash**, as given by argument zero.

_ Expands to the last argument to the previous command, after expansion. Also set to the full pathname of each command executed and placed in the environment exported to that command.

Shell Variables

The following variables are set by the shell:

PPID The process ID of the shell's parent.

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

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

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

UID Expands to the user ID of the current user, initialized at shell startup.

EUID Expands to the effective user ID of the current user, initialized at shell startup.

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

BASH_VERSION
Expands to the version number of this instance of **bash**.

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

RANDOM
Each time this parameter is referenced, a random integer is generated. The sequence of random numbers may be initialized by assigning a value to **RANDOM**. If **RANDOM** is unset, it loses its special properties, even if it is subsequently reset.

SECONDS
Each time this parameter is referenced, the number of seconds since shell invocation is returned. 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. If **SECONDS** is unset, it loses its special properties, even if it is subsequently reset.

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. When in a function, the value is not the number of the source line that the command appears on (that information has been lost by the time the function is executed), but is an approximation of the number of *simple commands* executed in the current function. If **LINENO** 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. If **HISTCMD** is unset, it loses its special properties, even if it is subsequently reset.

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).

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

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

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

IFS The *Internal Field Separator* that is used for word splitting after expansion and to split lines into words with the **read** builtin command. The default value is ‘‘<space><tab><newline>’’.

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). The default path is system−dependent, and is set by the administrator who installs **bash**. A common value is ‘‘/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:.’’.

HOME
The home directory of the current user; the default argument for the **cd** builtin command.

CDPATH
The search path for the **cd** command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the **cd** command. A sample value is ‘‘.:˜:/usr’’.

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

MAIL If this parameter is set to a filename and the **MAILPATH** variable is not set, **bash** informs the user of the arrival of mail in the specified file.

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 prompting. If this variable is unset, the shell disables mail checking.

MAILPATH
A colon-separated list of pathnames to be checked for mail. The message to be printed may be specified by separating the pathname from the message with a ‘?’. $_ stands for the name of the current mailfile. Example:
MAILPATH=’/usr/spool/mail/bfox?"You have mail":˜/shell-mail?"$_ has mail!"’
Bash supplies a default value for this variable, but the location of the user mail files that it uses is system dependent (e.g., /usr/spool/mail/**$USER**).

MAIL_WARNING
If set, and a file that **bash** is checking for mail has been accessed since the last time it was checked, the message ‘‘The mail in *mailfile* has been read’’ is printed.

PS1 The value of this parameter is expanded (see **PROMPTING** below) and used as the primary prompt string. The default value is ‘‘**bash\$** ’’.

PS2 The value of this parameter is expanded 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 and the value is printed before each command **bash** displays during an execution trace. The first character of **PS4** is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is ‘‘**+** ’’.

HISTSIZE
The number of commands to remember in the command history (see **HISTORY** below). The default value is 500.

HISTFILE
The name of the file in which command history is saved. (See **HISTORY** below.) The default value is *~/.bash_history*. If unset, the command history is not saved when an interactive shell 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 that number of lines. The default value is 500.

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.

PROMPT_COMMAND
If set, the value is executed as a command prior to issuing each primary prompt.

IGNOREEOF
Controls the action of the shell on receipt of an **EOF** character as the sole input. If set, the value is the number of consecutive **EOF** characters typed as the first characters on an input line before **bash** exits. If the variable exists but does not have a numeric value, or has no value, the default value is 10. If it does not exist, **EOF** signifies the end of input to the shell. This is only in effect for interactive shells.

TMOUT
If set to a value greater than zero, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt. **Bash** terminates after waiting for that number of seconds if input does not arrive.

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:~’’.

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

notify If set, **bash** reports terminated background jobs immediately, rather than waiting until before printing the next primary prompt (see also the **−b** option to the **set** builtin command).

history_control
HISTCONTROL
If set to a value of *ignorespace*, lines which begin with a **space** character are not entered on the history list. If set to a value of *ignoredups*, lines matching the last history line are not entered. A value of *ignoreboth* combines the two options. If unset, or if set to any other value than those above, all lines read by the parser are saved on the history list.

command_oriented_history
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.

glob_dot_filenames
If set, **bash** includes filenames beginning with a ‘.’ in the results of pathname expansion.

allow_null_glob_expansion
If set, **bash** allows pathname patterns which match no files (see **Pathname Expansion** below) to expand to a null string, rather than themselves.

histchars
The two or three characters which control history expansion and tokenization (see **HISTORY EXPANSION** below). The first character is the *history expansion character*, that is, the character which signals the start of a history expansion, normally ‘**!**’. The second character is the *quick substitution* character, which is used as shorthand for re-running the previous command entered,

substituting one string for another in the command. The default is '^'. The optional third character is the character which signifies that the remainder of the line is a comment, when found as the first character of a word, normally '**#**'. The history comment character causes history substitution to be skipped 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.

nolinks

If set, the shell does not follow symbolic links when executing commands 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, such as **cd**. See also the description of the **−P** option to the **set** builtin (**SHELL BUILTIN COMMANDS** below).

hostname_completion_file
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 file may be changed interactively; the next time hostname completion is attempted **bash** adds the contents of the new file to the already existing database.

noclobber

If set, **bash** does not overwrite an existing file with the **>**, **>&**, and **<>** redirection operators. This variable may be overridden when creating output files by using the redirection operator **>|** instead of **>** (see also the **−C** option to the **set** builtin command).

auto_resume

This variable controls how the shell interacts with the user and job control. If this variable is set, single word simple commands 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 the string typed, the job most recently accessed is selected. The *name* of a stopped job, in this context, is the command line used to start it. If set to the value *exact*, the string supplied must match the name of a stopped job exactly; if set to *substring*, the string supplied needs to match a substring of the name of a stopped job. The *substring* value provides functionality analogous to the **%?** job id (see **JOB CONTROL** below). If set to any other value, the supplied string must be a prefix of a stopped job's name; this provides functionality analogous to the **%** job id.

no_exit_on_failed_exec

If this variable exists, a non-interactive shell will not exit if it cannot execute the file specified in the **exec** builtin command. An interactive shell does not exit if **exec** fails.

cdable_vars

If this is 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.

EXPANSION

Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: *brace expansion*, *tilde expansion*, *parameter and variable expansion*, *command substitution*, *arithmetic expansion*, *word splitting*, and *pathname expansion*.

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

On systems that can support it, there is an additional expansion available: *process substitution*.

Only brace expansion, word splitting, and pathname expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The single exception to this is the expansion of "**$@**" as explained above (see **PARAMETERS**).

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *pathname expansion*, but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional *preamble*, followed by a series of comma-separated strings between a pair of braces, followed by an optional *postamble*. The preamble is prepended to each string contained within the braces, and the postamble 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; left to right order is preserved. For example, a**{**d,c,b**}**e expands into 'ade ace abe'.

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. Any incorrectly formed brace expansion is left unchanged.

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 traditional versions of **sh**, the Bourne shell. **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. The same word is output as *file1 file2* after expansion by **bash**. If strict compatibility with **sh** is desired, start **bash** with the **−nobraceexpansion** flag (see **OPTIONS** above) or disable brace expansion with the **+o braceexpand** option to the **set** command (see **SHELL BUILTIN COMMANDS** below).

Tilde Expansion

If a word begins with a tilde character ('~'), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible *login name*. If this *login name* is the null string, the tilde is replaced with the value of the parameter **HOME**. If **HOME** is unset, the home directory of the user executing the shell is substituted instead.

If a '+' follows the tilde, the value of **PWD** replaces the tilde and '+'. If a '−' follows, the value of **OLDPWD** is substituted. If the value following the tilde is a valid *login name*, the tilde and *login name* are replaced with the home directory associated with that name. If the name is invalid, or the tilde expansion fails, the word is unchanged.

Each variable assignment is checked for unquoted instances of tildes following a **:** or **=**. In these cases, tilde substitution is also performed. Consequently, one may use pathnames with tildes in assignments to **PATH**, **MAILPATH**, and **CDPATH**, and the shell assigns the expanded value.

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.

${*parameter*}
: The value of *parameter* is substituted. 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.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. **Bash** tests for a parameter that is unset or null; omitting the colon results in a test 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*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.

${*parameter*:?*word*}

Display Error if Null or Unset. If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. 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.

${#*parameter*}

The length in characters of the value of *parameter* is substituted. If *parameter* is * or @, the length substituted is the length of * expanded within double quotes.

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

The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches the beginning of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern deleted (the "#" case) or the longest matching pattern deleted (the "##" case).

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

The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches a trailing portion of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern deleted (the "%" case) or the longest matching pattern deleted (the "%%" case).

Command Substitution

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

$(*command***)**

or

‘*command***‘**

Bash performs the expansion by executing *command* and replacing the command substitution with the standard output of the command, with any trailing newlines deleted.

When the old−style backquote form of substitution is used, backslash retains its literal meaning except when followed by **$**, **‘**, or ****. When using the $(*command*) form, all characters between the parentheses make up the command; none are treated specially.

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

If the substitution appears within double quotes, word splitting and pathname expansion are not performed on the results.

Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. There are two formats for arithmetic expansion:

$[*expression***]**

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

The *expression* is treated as if it were within double quotes, but a double quote inside the braces or parentheses is not treated specially. All tokens in the expression undergo parameter expansion, command substitution, and quote removal. Arithmetic substitutions may be nested.

The evaluation is performed according to the rules listed below under **ARITHMETIC EVALUATION**. If *expression* is invalid, **bash** prints a message indicating failure and no substitution occurs.

Process Substitution

Process substitution is supported on systems that support named pipes (*FIFOs*) or the **/dev/fd** method of naming open files. It takes the form of **<(***list***)** or **>(***list***)**. The process *list* is run with its input or output connected to a *FIFO* or some file in **/dev/fd**. The name of this file 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 will provide input for *list*. If the **<(***list***)** form is used, the file passed as an argument should be read to obtain the output of *list*.

On systems that support it, *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*.

The shell treats each character of **IFS** as a delimiter, and splits the results of the other expansions into words on these characters. If the value of **IFS** is exactly **<space><tab><newline>**, the default, then any sequence of **IFS** characters serves to delimit words. If **IFS** has a value other than the default, then sequences of the whitespace characters **space** and **tab** are ignored at the beginning and end of the word, as long as the whitespace character is in the value of **IFS** (an **IFS** whitespace character). Any character in **IFS** that is not **IFS** whitespace, along with any adjacent **IFS** whitespace characters, delimits a field. A sequence of **IFS** whitespace characters is also treated as a delimiter. If the value of **IFS** is null, no word splitting occurs. **IFS** cannot be unset.

Explicit null arguments (**""** or **''**) are retained. Implicit null arguments, resulting from the expansion of *parameters* that have no values, are removed.

Note that if no expansion occurs, no splitting is performed.

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, then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of pathnames matching the pattern. If no matching pathnames are found, and the shell variable **allow_null_glob_expansion** is unset, the word is left unchanged. If the variable is set, and no matches are found, the word is removed. When a pattern is used for pathname generation, the character **"."** at the start of a name or immediately following a slash must be matched explicitly, unless the shell variable **glob_dot_filenames** is set. The slash character must always be matched explicitly. In other cases, the **"."** character is not treated specially.

The special pattern characters have the following meanings:

***** Matches any string, including the null string.

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a *range*; any character lexically between those two characters, inclusive, is matched. If the first character following the **[** is a **!** or a **^** then any character not enclosed is matched. A **–** or **]** may be matched by including it as the first or last character in the set.

Quote Removal

After the preceding expansions, all unquoted occurrences of the characters ****, **'**, and **"** are removed.

REDIRECTION

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection may also be used to open and close files for 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.

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

Note that 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 duplicated as standard output before the standard output was redirected to *dirlist*.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened 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

Redirection of output causes the file whose name results from the expansion of *word* to be opened 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 **>|**, then the value of the **-C** option to the **set** builtin command is not tested, and file creation is attempted. (See also the description of **noclobber** under **Shell Variables** above.)

Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of *word* to be opened 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

Bash allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word* with this construct.

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

Here Documents

This type of redirection instructs the shell to read input from the current source until a line containing only *word* (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command.

The format of here-documents is as follows:

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

No parameter expansion, command substitution, pathname expansion, or arithmetic expansion is performed on *word*. If any characters in *word* are quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. Otherwise, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion. In the latter case, the pair **\<newline>** is ignored, and **** must be used to quote the characters ****, **$**, and **‘**.

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

Duplicating File Descriptors

The redirection operator

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

is used to duplicate input file descriptors. If *word* expands to one or more digits, the file descriptor denoted by *n* is made to be a copy of that file descriptor. If *word* evaluates to **–**, file descriptor *n* is closed. If *n* is not specified, the standard input (file descriptor 0) is used.

The operator

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

is used similarly to duplicate output file descriptors. If *n* is not specified, the standard output (file descriptor 1) is used. As a special case, if *n* is omitted, and *word* does not expand to one or more digits, the standard output and standard error are redirected as described previously.

Opening File Descriptors for Reading and Writing

The redirection operator

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

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

FUNCTIONS

A shell function, defined as described above under **SHELL GRAMMAR**, stores a series of commands for later execution. Functions are executed in the context of the current shell; no new process is 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 change. Positional parameter 0 is unchanged.

Variables local to the function may be declared with the **local** builtin command. Ordinarily, variables and their values are shared between the function and its caller.

If the builtin command **return** is executed in a function, the function completes and execution resumes with the next command after the function call. When a function completes, the values of the positional parameters and the special parameter **#** are restored to the values they had prior to function execution.

Function names and definitions may be listed with the **–f** option to the **declare** or **typeset** builtin commands. Functions may be exported so that subshells automatically have them defined with the **–f** option to the **export** builtin.

Functions may be recursive. No limit is imposed on the number of recursive calls.

ALIASES

The shell maintains a list of *aliases* that may be set and unset with the **alias** and **unalias** builtin commands (see **SHELL BUILTIN COMMANDS** below). The first word of each command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The alias name and the replacement text may contain any valid shell input, including the *metacharacters* listed above, with the exception that the alias name may not contain =. 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*, then the next command word following the alias is also checked 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, as in **csh**. If arguments are needed, a shell function should be used.

Aliases are not expanded when the shell is not interactive.

The rules concerning the definition and use of aliases are somewhat confusing. **Bash** always reads at least one complete line of input before executing any of the commands on that line. 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 next line of input is read. This means that the commands following the alias definition on that line are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when the function definition is read, not when the function is executed, because a function definition is itself a compound 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.

Note that for almost every purpose, aliases are superseded by shell functions.

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 system's terminal driver and **bash**.

The shell associates a *job* with each pipeline. It keeps a table of currently executing jobs, which may be listed with the **jobs** command. 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, the system maintains the notion of a *current terminal process group ID*. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as **SIGINT**. These processes are said to be in the *foreground*. *Background* processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or write to the terminal. Background processes which attempt to read from (write to) the terminal are sent a **SIGTTIN (SIGTTOU**) signal by the terminal driver, which, unless caught, suspends the process.

If the operating system on which **bash** is running supports job control, **bash** allows you to use it. Typing the *suspend* character (typically **^Z**, Control-Z) while a process is running causes that process to be stopped and returns you to **bash**. Typing the *delayed suspend* character (typically **^Y**, Control-Y) causes the process to be stopped when it attempts to read input from the terminal, and control to be returned to **bash**. You may then manipulate 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. A **^Z** takes effect immediately, and has the additional side effect of causing pending output and typeahead to be discarded.

There are a number of ways to refer to a job in the shell. The character **%** introduces a job name. 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 stopped **ce** job. If a prefix matches more than one job, **bash** reports an error. Using **%?ce**, on the other hand, refers to any job containing the string **ce** in its command line. If the substring matches more than one job, **bash** reports an error. The symbols **%%** and **%+** refer to the shell's notion of the *current job*, which is the last job stopped while it was in the foreground. The *previous job* may be referenced using **%−**. In output pertaining to jobs (e.g., the output of the **jobs** command), the current job is always flagged 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 reporting changes in a job's status so as to not interrupt any other output. If the **-b** option to the **set** builtin command is set, **bash** reports such changes immediately. (See also the description of **notify** variable under **Shell Variables** above.)

If you attempt to exit **bash** while jobs are stopped, the shell prints a message warning you. You may then use the **jobs** command to inspect their status. If you do this, or try to exit again immediately, you are not warned again, and the stopped jobs are terminated.

SIGNALS

When **bash** is interactive, it ignores **SIGTERM** (so that **kill 0** does not kill an interactive shell), and **SIGINT** is caught and handled (so that the **wait** builtin is interruptible). In all cases, **bash** ignores **SIGQUIT**. If job control is in effect, **bash** ignores **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

Synchronous jobs started by **bash** have signals set to the values inherited by the shell from its parent. When job control is not in effect, background jobs (jobs started with **&**) ignore **SIGINT** and **SIGQUIT**. Commands run as a result of command substitution ignore the keyboard-generated job control signals **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

COMMAND EXECUTION

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

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. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.

If the search is successful, or if the command name contains one or more slashes, the shell executes the named program. 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. A subshell is spawned to execute it. 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.

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 allows you to manipulate the environment in several ways. 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** and **declare −x** commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value 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** command, plus any additions via the **export** and

declare −x commands.

The environment for any *simple command* or function may be augmented temporarily by prefixing it with parameter assignments, as described above in **PARAMETERS**. These assignment statements affect only the environment seen by that command.

If the **−k** flag 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 path name of the command and passed to that command in its environment.

EXIT STATUS

For the purposes of the shell, a command which exits with a zero exit status has succeeded. An exit status of zero indicates success. A non−zero exit status indicates failure. When a command terminates on a fatal signal, **bash** uses the value of 128+**signal** 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.

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.

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** allows these prompt strings to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

\t	the current time in HH:MM:SS format
\d	the date in "Weekday Month Date" format (e.g., "Tue May 26")
\n	newline
\s	the name of the shell, the basename of **$0** (the portion following the final slash)
\w	the current working directory
\W	the basename of the current working directory
\u	the username of the current user
\h	the hostname
\#	the command number of this command
\!	the history number of this command
\$	if the effective UID is 0, a **#**, otherwise a **$**
\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 word splitting.

READLINE

This is the library that handles reading input when using an interactive shell, unless the **−nolineediting** option is given. By default, the line editing commands are similar to those of emacs. A vi-style line editing interface is also available.

In this section, the emacs-style notation is used to denote 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. (On keyboards without a *meta* key, M−*x* means ESC *x*, i.e., press the Escape key then the *x* key. This makes ESC the *meta prefix*. The combination M−C−*x* means ESC−Control−*x*, or press the Escape key then hold the Control key while pressing the *x* key.)

The default key-bindings may be changed with an *~/.inputrc* file. The value of the shell variable **INPUTRC**, if set, is used instead of *~/.inputrc*. Other programs that use this library 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*.

The following symbolic character names are recognized: *RUBOUT*, *DEL*, *ESC*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *SPC*, *SPACE*, 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*).

Readline is customized by putting commands in an initialization file. The name of this file is taken from the value of the **INPUTRC** variable. If that variable is unset, the default is *~/.inputrc*. When a program which uses the readline library starts up, the init file is read, and the key bindings and variables are set. There are only a few basic constructs allowed in the readline init 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 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 name may be specified in one of two ways: as a symbolic key name, possibly with *Meta-* or *Control-* prefixes, or as a key sequence. 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.

```
"\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 escape sequences is

\C− control prefix

\M- meta prefix

\e an escape character

**** backslash

\" literal "

\' literal '

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. Backslash will quote any character in the macro text, including " and '.

Bash allows the current readline key bindings to be displayed or modified with the **bind** builtin command. The editing mode may be switched during interactive use by using the **−o** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below).

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*

Except where noted, readline variables can take the values **On** or **Off**. The variables and their default values are:

horizontal−scroll−mode (Off)
When set 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.

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

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

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.

comment−begin (“#”)
The string that is inserted in **vi** mode when the **vi−comment** command is executed.

meta−flag (Off)
If set to **On**, readline will enable eight-bit input (that is, it will not strip the high bit from the characters it reads), regardless of what the terminal claims it can support.

convert−meta (On)
If set to **On**, readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prepending an escape character (in effect, using escape as the *meta prefix*).

output−meta (Off)
If set to **On**, readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence.

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, the user is asked whether or not he wishes to view them; otherwise they are simply listed on the terminal.

keymap (emacs)
Set the current readline keymap. The set of legal keymap names is *emacs, emacs-standard, emacs-meta, emacs-ctlx, vi, vi-move, 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.

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.

expand−tilde (Off)
If set to **on**, tilde expansion is performed when readline attempts word completion.

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 three parser directives used.

$if The **$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 extends to the end of the line; 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 the full name of the terminal and the portion of the terminal name before the first **−**. This allows *sun* to match both *sun* and *sun−cmd*, for instance.

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

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

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

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**) causes that command to act in a backward direction. Commands whose behavior with arguments deviates from this are noted.

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 cause the text to be accumulated 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.

The following is a list of the names of the commands and the default key sequences to which they are bound.

Commands for Moving

beginning−of−line (C−a)
Move to the start of the current line.

end−of−line (C−e)
Move to the end of the line.

forward−char (C−f)
Move forward a character.

backward−char (C−b)
Move back a character.

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 this, or the previous, word. Words are composed of alphanumeric characters (letters and digits).

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

redraw−current−line
Refresh the current line. By default, this is unbound.

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** variable. If the line is a modified history line, then 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.

next−history (C−n)
Fetch the next command from the history list, moving forward in the list.

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.

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.

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.

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.

non−incremental−forward−search−history (M−n)
Search forward through the history using a non−incremental search for a string supplied by the user.

history−search−forward
Search forward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

history−search−backward
Search backward through the history for the string of characters between the start of the current line and the current point. This is a non-incremental search. By default, this command is unbound.

yank−nth−arg (M−C−y)
Insert the first argument to the previous command (usually the second word on the previous line) at point (the current cursor position). 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.

yank−last−arg (M−., M−_)
Insert the last argument to the previous command (the last word on the previous line). With an argument, behave exactly like **yank-nth-arg**.

shell−expand−line (M−C−e)
Expand the line the way the shell does when it reads it. This performs alias and history expansion as well as all of the shell word expansions. 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.

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

operate-and-get-next (C−o)
Accept the current line for execution and fetch the next line relative to the current line from the history for editing. Any argument is ignored.

Commands for Changing Text

delete−char (C−d)
Delete the character under the cursor. If point is at the beginning of the line, there are no characters in the line, and the last character typed was not **C−d**, then return **EOF**.

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

quoted−insert (C−q, C−v)
Add the next character that you type 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.

transpose−chars (C−t)
Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative arguments don't work.

transpose−words (M−t)
Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

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

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

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

Killing and Yanking

kill−line (C−k)
Kill the text from the current cursor position to the end of the line.

backward−kill−line (C−x C−Rubout)
Kill backward to the beginning of the line.

unix−line−discard (C−u)
Kill backward from point to the beginning of the line.

kill−whole−line
Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

kill−word (M−d)
Kill from the cursor 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 the cursor. Word boundaries are the same as those used by **backward−word**.

unix−word−rubout (C−w)
Kill the word behind the cursor, using white space as a word boundary. The word boundaries are different from backward−kill−word.

delete−horizontal−space
Delete all spaces and tabs around point. By default, this is unbound.

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

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
Each time this is executed, the argument count is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four. By default, this is not bound to a key.

Completing

complete (TAB)
Attempt to perform completion on the text before point. **Bash** attempts completion 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 and functions) in turn. If none of these produces a match, filename completion is attempted.

possible−completions (M-?)
List the possible completions of the text before point.

insert−completions
Insert all completions of the text before point that would have been generated by **possible−completions**. By default, this is not bound to a key.

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, 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 lines from the history list for possible completion matches.

complete−into−braces (M−{)
Perform filename completion and return 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 save 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.

Miscellaneous

re−read−init−file (C−x C−r)
: Read in the contents of your init 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−uppercase−version (M−a, M−b, ...)
: Run the command that is bound to the corresponding uppercase character.

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 typing the **undo** command enough times to return the line to its initial state.

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

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.

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

HISTORY

When interactive, the shell provides access to the *command history*, the list of commands previously typed. The text of the last **HISTSIZE** commands (default 500) is saved in a history list. 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 **command_oriented_history** and **HISTCONTROL**. On startup, the history is initialized from the file named by the variable **HISTFILE** (default *˜/.bash_history*). **HISTFILE** is truncated, if necessary, to contain no more than **HISTFILESIZE** lines. The builtin command **fc** (see **SHELL BUILTIN COMMANDS** below) may be used to list or edit and re-execute a portion of the history list. The **history** builtin can be used to display the history list and manipulate the history file. When using the command-line editing, search commands are available in each editing mode that provide access to the history list. When an interactive shell exits, the last **HISTSIZE** lines are copied from the history list to **HISTFILE**. If **HISTFILE** is unset, or if the history file is unwritable, the history is not saved.

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. This feature 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.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is to determine which line from the previous history to use during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is the *event*, and the portions of that line that are acted upon are *words*.

The line is broken into words in the same fashion as when reading input, so that several *metacharacter*–separated words surrounded by quotes are considered as one word. Only backslash (\) and single quotes can quote the history escape character, which is **!** by default.

The shell allows control of the various characters used by the history expansion mechanism (see the description of **histchars** above under **Shell Variables**).

Event Designators

An event designator is a reference to a command line entry in the history list.

! Start a history substitution, except when followed by a **blank**, newline, = or (.

!! Refer to the previous command. This is a synonym for '!−1'.

!*n* Refer to command line *n*.

!−*n* Refer to the current command line minus *n*.

!*string* Refer to the most recent command starting with *string*.

!?*string***[?]**
Refer to the most recent command containing *string*.

^*string1***^***string2***^**
Quick substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to "!!:s/*string1*/*string2*/" (see **Modifiers** below).

!# The entire command line typed so far.

Word Designators

A **:** separates the event specification from the word designator. It can 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 a 0 (zero).

0 (zero)
The zeroth word. For the shell, this is the command word.

n The *n*th word.

^ The first argument. That is, word 1.

$ The last argument.

% The word matched by the most recent '?*string*?' search.

*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; the empty string is returned in that case.

x* Abbreviates *x−$*.

x− Abbreviates *x−$* like **x***, but omits the last word.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a ':'.

h Remove a trailing pathname component, leaving only the head.

r Remove a trailing suffix of the form *.xxx*, leaving the basename.

e Remove all but the trailing suffix.

t Remove all leading pathname components, leaving the tail.

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.

s/*old***/***new***/**
Substitute *new* for the first occurrence of *old* in the event line. Any delimiter can be used in place of /. The final delimiter is optional if it is the last character of the event line. The delimiter may be quoted in *old* and *new* with a single backslash. If & appears in *new*, it is replaced by *old*. A single backslash will quote the &.

& 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.

ARITHMETIC EVALUATION

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the **let** builtin command and **Arithmetic Expansion**). Evaluation is done in long integers with no check for overflow, though division by 0 is trapped and flagged as an error. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

− + unary minus and plus
! ˜ logical and bitwise negation
*** / %** multiplication, division, remainder
+ − addition, subtraction
<< >> left and right bitwise shifts
<= >= < >
comparison
== != equality and inequality
& bitwise AND
ˆ bitwise exclusive OR
| bitwise OR
&& logical AND
|| logical OR
= *= /= %= += −= <<= >>= &= ˆ= |=
assignment

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. The value of a parameter is coerced to a long integer within an expression. A shell variable need not have its integer attribute turned on to be used in an expression.

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 *base* is a decimal number between 2 and 36 representing the arithmetic base, and *n* is a number in that base. If *base* is omitted, then base 10 is used.

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

SHELL BUILTIN COMMANDS

: [*arguments*]
No effect; the command does nothing beyond expanding *arguments* and performing any specified redirections. A zero exit code is returned.

. *filename* [*arguments*]
source *filename* [*arguments*]
Read and execute commands from *filename* in the current shell environment and return the exit status of the last command executed from *filename*. If *filename* does not contain a slash, pathnames in **PATH** are used to find the directory containing *filename*. The file searched for in **PATH** need not be executable. The current directory is searched if no file is found in **PATH**. If any *arguments* are supplied, they become the positional parameters when *file* is executed. Otherwise the positional parameters are unchanged. The return status is the status of the last command exited within the script (0 if no commands are executed), and false if *filename* is not found.

alias [*name*[=*value*] ...]
Alias with no arguments prints the list of aliases in the form *name*=*value* on standard output. When arguments are supplied, an alias is defined 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. For each *name* in the argument list for which no *value* is supplied, the name and value of the alias is printed. **Alias** returns true unless a *name* is given for which no alias has been defined.

bg [*jobspec*]
Place *jobspec* in the background, as if it had been started with **&**. If *jobspec* is not present, the shell's notion of the *current job* is used. **bg** *jobspec* returns 0 unless run when job control is

disabled or, when run with job control enabled, if *jobspec* was not found or started without job control.

bind [**−m** *keymap*] [**−lvd**] [**-q** *name*]
bind [**−m** *keymap*] **-f** *filename*
bind [**−m** *keymap*] *keyseq*:*function-name*

Display current **readline** key and function bindings, or bind a key sequence to a **readline** function or macro. The binding syntax accepted is identical to that of *.inputrc*, but each binding must be passed as a separate argument; e.g., '"\C-x\C-r": re−read−init−file'. 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*; *emacs* is equivalent to *emacs-standard*.

−l List the names of all **readline** functions

−v List current function names and bindings

−d Dump function names and bindings in such a way that they can be re-read

−f *filename*
Read key bindings from *filename*

−q *function*
Query about which keys invoke the named *function*

The return value is 0 unless an unrecognized option is given or an error occurred.

break [*n*]
Exit from within a **for**, **while**, or **until** loop. If *n* is specified, break *n* levels. *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 the shell is not executing a loop when **break** is executed.

builtin *shell−builtin* [*arguments*]
Execute the specified shell builtin, passing it *arguments*, and return its exit status. This is useful when you wish to define a function whose name is the same as a shell builtin, but need the functionality of the builtin within the function itself. The **cd** builtin is commonly redefined this way. The return status is false if *shell−builtin* is not a shell builtin command.

cd [*dir*] Change the current directory to *dir*. The variable **HOME** is the default *dir*. The variable **CDPATH** defines the search path for the directory containing *dir*. Alternative directory names are separated by a colon (:). A null directory name in **CDPATH** is the same as the current directory, i.e., "**.**". If *dir* begins with a slash (/), then **CDPATH** is not used. An argument of **−** is equivalent to **$OLDPWD**. The return value is true if the directory was successfully changed; false otherwise.

command [**-pVv**] *command* [*arg* ...]
Run *command* with *args* suppressing the normal shell function lookup. Only builtin commands or commands found in the **PATH** are executed. If the **−p** option is given, 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, a description of *command* is printed. The **−v** option causes a single word indicating the command or pathname used to invoke *command* to be printed; the **−V** option produces a more verbose description. An argument of **−−** disables option checking for the rest of the arguments. If the **−V** or **−v** option is supplied, the exit status is 0 if *command* was found, and 1 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*.

continue [*n*]
Resume the next iteration of the enclosing **for**, **while**, or **until** loop. If *n* is specified, resume at the *n*th enclosing loop. *n* must be ≥ 1. If *n* is greater than the number of enclosing loops, the last enclosing loop (the 'top−level' loop) is resumed. The return value is 0 unless the shell is not executing a loop when **continue** is executed.

declare [**−frxi**] [*name*[=*value*]]
typeset [**−frxi**] [*name*[=*value*]]

Declare variables and/or give them attributes. If no *name*s are given, then display the values of variables instead. The options can be used to restrict output to variables with the specified attribute.

−f Use function names only
−r Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements.
−x Mark *name*s for export to subsequent commands via the environment.
−i The variable is treated as an integer; arithmetic evaluation (see **ARITHMETIC EVALUATION**) is performed when the variable is assigned a value.

Using '+' instead of '−' turns off the attribute instead. When used in a function, makes *name*s local, as with the **local** command. The return value is 0 unless an illegal option is encountered, an attempt is made to define a function using "-f foo=bar", one of the *names* is not a legal shell variable name, an attempt is made to turn off readonly status for a readonly variable, or an attempt is made to display a non-existant function with -f.

dirs [-l] [+/−n]

Display the list of currently remembered directories. Directories are added to the list with the **pushd** command; the **popd** command moves back up through the list.

+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.
−l produces a longer listing; the default listing format uses a tilde to denote the home directory.

The return value is 0 unless an illegal option is supplied or *n* indexes beyond the end of the directory stack.

echo [**−neE**] [*arg* ...]

Output the *arg*s, separated by spaces. The return status is always 0. If **−n** is specified, the trailing newline is suppressed. If the **−e** option is given, interpretation of the following backslash-escaped characters is enabled. The **−E** option disables the interpretation of these escape characters, even on systems where they are interpreted by default.

\a alert (bell)
\b backspace
\c suppress trailing newline
\f form feed
\n new line
\r carriage return
\t horizontal tab
\v vertical tab
**** backslash
\nnn the character whose ASCII code is *nnn* (octal)

enable [**−n**] [**−all**] [*name* ...]

Enable and disable builtin shell commands. This allows the execution of a disk command which has the same name as a shell builtin without specifying a full pathname. If **−n** is used, each *name* is disabled; otherwise, *names* are enabled. For example, to use the **test** binary found via the **PATH** instead of the shell builtin version, type ''enable -n test''. If no arguments are given, a list of all enabled shell builtins is printed. If only **−n** is supplied, a list of all disabled builtins is printed. If only **−all** is supplied, the list printed includes all builtins, with an indication of whether or not each is enabled. **enable** accepts **−a** as a synonym for **−all**. The return value is 0 unless a *name* is not a shell builtin.

eval [*arg* ...]
The *arg*s are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of the **eval** command. If there are no *args*, or only null arguments, **eval** returns true.

exec [[**−**] *command* [*arguments*]]
If *command* is specified, it replaces the shell. No new process is created. The *arguments* become the arguments to *command*. If the first argument is **−**, the shell places a dash in the zeroth arg passed to *command*. This is what login does. If the file cannot be executed for some reason, a non-interactive shell exits, unless the shell variable **no_exit_on_failed_exec** exists, in which case it returns failure. An interactive shell returns failure if the file cannot be executed. If *command* is not specified, any redirections take effect in the current shell, and the return status is 0.

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. A trap on **EXIT** is executed before the shell terminates.

export [**−nf**] [*name*[=*word*]] ...
export −p
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. If no *names* are given, or if the **−p** option is supplied, a list of all names that are exported in this shell is printed. The **−n** option causes the export property to be removed from the named variables. An argument of **−−** disables option checking for the rest of the arguments. **export** returns an exit status of 0 unless an illegal option is encountered, one of the *names* is not a legal shell variable name, or **−f** is supplied with a *name* that is not a function.

fc [**−e** *ename*] [**−nlr**] [*first*] [*last*]
fc −s [*pat*=*rep*] [*cmd*]
Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. *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). 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.

The **−n** flag suppresses the command numbers when listing. The **−r** flag reverses the order of the commands. If the **−l** flag is given, the commands are listed on standard output. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the **FCEDIT** variable is used, and the value of **EDITOR** if **FCEDIT** is not set. If neither variable is set, *vi* is used. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after each instance of *pat* is replaced by *rep*. A useful alias to use with this 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 0 unless an illegal option is encountered or *first* or *last* specify history lines out of range. If the **−e** option is supplied, the return value is the value of the last command executed or failure if an error occurs with the temporary file of commands. If the second form is used, the return status is that of the command re-executed, unless *cmd* does not specify a valid history line, in which case **fc** returns failure.

fg [*jobspec*]
Place *jobspec* in the foreground, and make it the current job. If *jobspec* is not present, the shell's notion of the *current job* is used. 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* [*args*]

getopts is used by shell procedures to parse positional parameters. *optstring* contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. 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 if a new set of parameters is to be used.

getopts can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation diagnostic messages are printed when illegal options or missing option arguments are encountered. If the variable **OPTERR** is set to 0, no error message will be displayed, even if the first character of *optstring* is not a colon.

If an illegal option is seen, **getopts** places ? into *name* and, if not silent, prints an error message and unsets **OPTARG**. If **getopts** is silent, the option character found is placed in **OPTARG** and no diagnostic message is printed.

If a required argument is not found, and **getopts** is not silent, a question mark (**?**) is placed in *name*, **OPTARG** is unset, and a diagnostic message is printed. If **getopts** is silent, then a colon (**:**) is placed in *name* and **OPTARG** is set to the option character found.

getopts normally parses the positional parameters, but if more arguments are given in *args*, **getopts** parses those instead. **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 [**−r**] [*name*]

For each *name*, the full pathname of the command is determined and remembered. The **−r** option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is printed. An argument of **−−** disables option checking for the rest of the arguments. The return status is true unless a *name* is not found or an illegal option is supplied.

help [*pattern*]

Display helpful information about builtin commands. If *pattern* is specified, **help** gives detailed help on all commands matching *pattern*; otherwise a list of the builtins is printed. The return status is 0 unless no command matches *pattern*.

history [*n*]
history −rwan [*filename*]

With no options, display the command history list with line numbers. Lines listed with a ***** have been modified. An argument of *n* lists only the last *n* lines. If a non-option argument is supplied, it is used as the name of the history file; if not, the value of **HISTFILE** is used. Options, if supplied, have the following meanings:

- **−a** Append the ‘‘new’’ history lines (history lines entered since the beginning of the current **bash** session) to the history file
- **−n** Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current **bash** session.
- **−r** Read the contents of the history file and use them as the current history
- **−w** Write the current history to the history file, overwriting the history file’s contents.

The return value is 0 unless an illegal option is encountered or an error occurs while reading or writing the history file.

jobs [**−lnp**] [*jobspec* ...]
jobs −x *command* [*args* ...]

The first form lists the active jobs. The **−l** option lists process IDs in addition to the normal information; the **−p** option lists only the process ID of the job's process group leader. The **−n** option displays only jobs that have changed status since last notified. If *jobspec* is given, output is restricted to information about that job. The return status is 0 unless an illegal option is encountered or an illegal *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** | **−sigspec**] [*pid* | *jobspec*] ...
kill −l [*signum*]

Send the signal named by *sigspec* to the processes named by *pid* or *jobspec*. *sigspec* is either a signal name such as **SIGKILL** or a signal number. If *sigspec* is a signal name, the name is case insensitive and may be given with or without the **SIG** prefix. If *sigspec* is not present, then **SIGTERM** is assumed. An argument of **−l** lists the signal names. If any arguments are supplied when **−l** is given, the names of the specified signals are listed, and the return status is 0. An argument of **−−** disables option checking for the rest of the arguments. **kill** returns true if at least one signal was successfully sent, or false if an error occurs or an illegal option is encountered.

let *arg* [*arg* ...]

Each *arg* is an arithmetic expression to be evaluated (see **ARITHMETIC EVALUATION**). If the last *arg* evaluates to 0, **let** returns 1; 0 is returned otherwise.

local [*name*[=*value*] ...]

For each argument, create a local variable named *name*, and assign it *value*. When **local** is used within a function, it causes the variable *name* to have a visible scope restricted to that function and its children. With no operands, **local** writes a list of local variables to the standard output. It is an error to use **local** when not within a function. The return status is 0 unless **local** is used outside a function, or an illegal *name* is supplied.

logout Exit a login shell.

popd [**+/−n**]

Removes entries from the directory stack. With no arguments, removes the top directory from the stack, and performs a **cd** to the new top directory.

+n removes the *n*th entry counting from the left of the list shown by **dirs**, starting with zero. For example: ‘‘popd +0’’ removes the first directory, ‘‘popd +1’’ the second.

−n removes 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 **popd** command is successful, a **dirs** is performed as well, and the return status is 0. **popd** returns false if an illegal option is encountered, the directory stack is empty, a non-existent directory stack entry is specified, or the directory change fails.

pushd [*dir*]
pushd +/−n

Adds a directory to the top of the directory stack, or rotates the stack, making the new top of the stack the current working directory. With no arguments, exchanges the top two directories and returns 0, unless the directory stack is empty.

+n Rotates the stack so that the *n*th directory (counting from the left of the list shown by **dirs**) is at the top.

−n Rotates the stack so that the *n*th directory (counting from the right) is at the top.

dir adds *dir* to the directory stack at the top, making it the new current working directory.

If the **pushd** command is successful, a **dirs** is performed as well. If the first form is used, **pushd** returns 0 unless the cd to *dir* fails. With the second form, **pushd** returns 0 unless the directory stack is empty, a non-existant directory stack element is specified, or the directory change to the specified new current directory fails.

pwd Print the absolute pathname of the current working directory. The path printed contains no symbolic links if the **−P** option to the **set** builtin command is set. See also the description of **nolinks** under **Shell Variables** above). The return status is 0 unless an error occurs while reading the pathname of the current directory.

read [**−r**] [*name* ...]
One line is read from the standard input, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. Only the characters in **IFS** are recognized as word delimiters. If no *names* are supplied, the line read is assigned to the variable **REPLY**. The return code is zero, unless end-of-file is encountered. If the **−r** option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line.

readonly [**−f**] [*name* ...]
readonly -p
The given *names* are marked readonly and the values of these *names* may not be changed by subsequent assignment. If the **−f** option is supplied, the functions corresponding to the *names* are so marked. If no arguments are given, or if the **−p** option is supplied, a list of all readonly names is printed. An argument of **−−** disables option checking for the rest of the arguments. The return status is 0 unless an illegal option is encountered, one of the *names* is not a legal shell variable name, or **−f** is supplied with a *name* that is not a function.

return [*n*]
Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script 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 used outside a function and not during execution of a script by **.**, the return status is false.

set [**−−abefhkmnptuvxldCHP**] [**-o** *option*] [*arg* ...]

−a Automatically mark variables which are modified or created for export to the environment of subsequent commands.

−b Cause the status of terminated background jobs to be reported immediately, rather than before the next primary prompt. (Also see **notify** under **Shell Variables** above).

−e Exit immediately if a *simple-command* (see **SHELL GRAMMAR** above) exits with a non−zero status. The shell does not exit if the command that fails is part of an *until* or *while* loop, part of an *if* statement, part of a **&&** or **||** list, or if the command's return value is being inverted via **!**.

−f Disable pathname expansion.

−h Locate and remember function commands as functions are defined. Function commands are normally looked up when the function is executed.

−k All keyword arguments are placed in the environment for a command, not just those that precede the command name.

−m Monitor mode. Job control is enabled. This flag is on by default for interactive shells on systems that support it (see **JOB CONTROL** above). Background processes run in a separate process group and a line containing their exit status is printed upon their completion.

−n Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored for interactive shells.

−o *option-name*
The *option-name* can be one of the following:
allexport
Same as **−a**.

braceexpand
The shell performs brace expansion (see **Brace Expansion** above). This is on by default.

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 **−nolineediting** option.

errexit Same as **−e**.

histexpand
Same as **−H**.

ignoreeof
The effect is as if the shell command 'IGNOREEOF=10' had been executed (see **Shell Variables** above).

interactive−comments
Allow a word beginning with **#** to cause that word and all remaining characters on that line to be ignored in an interactive shell (see **COMMENTS** above).

monitor Same as **−m**.

noclobber
Same as **−C**.

noexec Same as **−n**.

noglob Same as **−f**.

nohash Same as **−d**.

notify Same as **−b**.

nounset Same as **−u**.

physical Same as **−P**.

posix Change the behavior of bash where the default operation differs from the Posix 1003.2 standard to match the standard.

privileged
Same as **−p**.

verbose Same as **−v**.

vi Use a vi-style command line editing interface.

xtrace Same as **−x**.

If no *option-name* is supplied, the values of the current options are printed.

−p Turn on *privileged* mode. In this mode, the **$ENV** file is not processed, and shell functions are not inherited from the environment. This is enabled automatically on startup if the effective user (group) id is not equal to the real user (group) id. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

−t Exit after reading and executing one command.

−u Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, 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*, **bash** displays the expanded value of **PS4**, followed by the command and its expanded arguments.

−l Save and restore the binding of *name* in a **for** *name* [in **word**] command (see **SHELL GRAMMAR** above).

−d Disable the hashing of commands that are looked up for execution. Normally, commands are remembered in a hash table, and once found, do not have to be looked up again.

−C The effect is as if the shell command 'noclobber=' had been executed (see **Shell Variables** above).

−H Enable **!** style history substitution. This flag is on by default when the shell is interactive.

−P If set, do not follow symbolic links when performing commands such as **cd** which change the current directory. The physical directory is used instead.

−− If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the *arg*s, even if some of them begin with a **−**.

− Signal the end of options, cause all remaining *arg*s to be assigned to the positional parameters. The **−x** and **−v** options are turned off. If there are no *arg*s, the positional parameters remain unchanged.

The flags are off by default unless otherwise noted. Using + rather than − causes these flags to be turned off. The flags can also be specified as options to an invocation of the shell. The current set of flags may be found in **$−**. After the option arguments are processed, the remaining *n arg*s are treated as values for the positional parameters and are assigned, in order, to **$1**, **$2**, **...** **$***n*. If no options or *arg*s are supplied, all shell variables are printed. The return status is always true unless an illegal option is encountered.

shift [*n*]

The positional parameters from *n*+1 ... are renamed to **$1 ....** Parameters represented by the numbers **$#** down to **$#**−*n*+1 are unset. If *n* is 0, no parameters are changed. If *n* is not given, it is assumed to be 1. *n* must be a non-negative number less than or equal to **$#**. If *n* is greater than **$#**, the positional parameters are not changed. The return status is greater than 0 if *n* is greater than **$#** or less than 0; otherwise 0.

suspend [**−f**]

Suspend the execution of this shell until it receives a **SIGCONT** signal. The **−f** option says not to complain if this is a login shell; just suspend anyway. The return status is 0 unless the shell is a login shell and **−f** is not supplied, or if job control is not enabled.

test *expr*

[*expr* **]** Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression *expr*. Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Each operator and operand must be a separate argument. If *file* is of the form /dev/fd/*n*, then file descriptor *n* is checked.

−b *file* True if *file* exists and is block special.
−c *file* True if *file* exists and is character special.
−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.
−k *file* True if *file* has its ‘‘sticky’’ bit set.
−L *file* True if *file* exists and is a symbolic link.
−p *file* True if *file* exists and is a named pipe.
−r *file* True if *file* exists and is readable.
−s *file* True if *file* exists and has a size greater than zero.
−S *file* True if *file* exists and is a socket.
−t *fd* True if *fd* is opened on 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.
−O *file* True if *file* exists and is owned by the effective user id.
−G *file* True if *file* exists and is owned by the effective group id.

file1 **−nt** *file2*
True if *file1* is newer (according to modification date) than *file2*.

file1 **−ot** *file2*
True if *file1* is older than file2.

file1 **−ef** *file*
True if *file1* and *file2* have the same device and inode numbers.

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

−n *string*

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

string1 **=** *string2*
True if the strings are equal.

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

! *expr* True if *expr* is false.

expr1 **−a** *expr2*
True if both *expr1* AND *expr2* are true.

expr1 **−o** *expr2*
True if either *expr1* OR *expr2* is true.

arg1 **OP** *arg2*
OP is one of **−eq**, **−ne**, **−lt**, **−le**, **−gt**, or **−ge**. These arithmetic binary operators return true if *arg1* is equal, not-equal, less-than, less-than-or-equal, greater-than, or greater-than-or-equal than *arg2*, respectively. *Arg1* and *arg2* may be positive integers, negative integers, or the special expression **−l** *string*, which evaluates to the length of *string*.

times Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.

trap [**−l**] [*arg*] [*sigspec*]
The command *arg* is to be read and executed when the shell receives signal(s) *sigspec*. If *arg* is absent or **−**, all specified signals are reset to their original values (the values they had upon entrance to the shell). If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. *sigspec* is either a signal name defined in <*signal.h*>, or a signal number. If *sigspec* is **EXIT** (0) the command *arg* is executed on exit from the shell. With no arguments, **trap** prints the list of commands associated with each signal number. The **−l** option causes the shell to print a list of signal names and their corresponding numbers. An argument of **−−** disables option checking for the rest of the arguments. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if either the trap name or number is invalid; otherwise **trap** returns true.

type [**−all**] [**−type** | **−path**] *name* [*name* ...]
With no options, indicate how each *name* would be interpreted if used as a command name. If the **−type** flag is used, **type** prints a phrase which is one of *alias*, *keyword*, *function*, *builtin*, or *file* if *name* is an alias, shell reserved word, function, builtin, or disk file, respectively. If the name is not found, then nothing is printed, and an exit status of false is returned. If the **−path** flag is used, **type** either returns the name of the disk file that would be executed if *name* were specified as a command name, or nothing if **−type** would not return *file*. If a command is hashed, **−path** prints the hashed value, not necessarily the file that appears first in **PATH**. If the **−all** flag is used, **type** prints all of the places that contain an executable named *name*. This includes aliases and functions, if and only if the **−path** flag is not also used. The table of hashed commands is not consulted when using **−all**. **type** accepts **−a**, **−t**, and **−p** in place of **−all**, **−type**, and **−path**, respectively. An argument of **−−** disables option checking for the rest of the arguments. **type** returns true if any of the arguments are found, false if none are found.

ulimit [**−SHacdfmstpnuv** [*limit*]]
Ulimit provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The value of *limit* can be a number in the unit specified for the resource, or the value **unlimited**. The **H** and **S** options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased 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, the command applies to the soft limit. If *limit* is omitted, the current value of the soft limit of the resource is printed, unless the **H** option is given. When more than one resource is specified, the limit name and unit is printed before the value. Other options are interpreted as follows:

–a	all current limits are reported
–c	the maximum size of core files created
–d	the maximum size of a process's data segment
–f	the maximum size of files created by the shell
–m	the maximum resident set size
–s	the maximum stack size
–t	the maximum amount of cpu time in seconds
–p	the pipe size in 512-byte blocks (this may not be set)
–n	the maximum number of open file descriptors (most systems do not allow this value to be set, only displayed)
–u	the maximum number of processes available to a single user
–v	The maximum amount of virtual memory available to the shell

An argument of **––** disables option checking for the rest of the arguments. If *limit* is given, it is the new value of the specified resource (the **–a** option is display only). If no option is given, then **–f** is assumed. Values are in 1024-byte increments, except for **–t**, which is in seconds, **–p**, which is in units of 512-byte blocks, and **–n** and **–u**, which are unscaled values. The return status is 0 unless an illegal option is encountered, a non-numeric argument other than **unlimited** is supplied as *limit*, or an error occurs while setting a new limit.

umask [**–S**] [*mode*]
The user file-creation mask is set 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, or if the **–S** option is supplied, the current value of the mask is printed. The **–S** option causes the mask to be printed in symbolic form; the default output is an octal number. An argument of **––** disables option checking for the rest of the arguments. The return status is 0 if the mode was successfully changed or if no *mode* argument was supplied, and false otherwise.

unalias [–**a**] [*name* ...]
Remove *name*s from the list of defined aliases. If **–a** is supplied, all alias definitions are removed. The return value is true unless a supplied *name* is not a defined alias.

unset [–**fv**] [*name* ...]
For each *name*, remove the corresponding variable or, given the **–f** option, function. An argument of **––** disables option checking for the rest of the arguments. Note that **PATH**, **IFS**, **PPID**, **PS1**, **PS2**, **UID**, and **EUID** cannot be unset. If any of **RANDOM**, **SECONDS**, **LINENO**, or **HISTCMD** are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a *name* does not exist or is non-unsettable.

wait [*n*]
Wait for the specified process and return its termination status. *n* may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If *n* is not given, all currently active child processes are waited for, and the return status is zero. If *n* specifies a non-existant process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

INVOCATION

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

An *interactive* shell is one whose standard input and output are both connected to terminals (as determined by *isatty*(3)), or one started with the **–i** option. **PS1** is set and **$–** includes **i** if **bash** is interactive, allowing a shell script or a startup file to test this state.

Login shells:
  On login (subject to the **–noprofile** option):
    if */etc/profile* exists, source it.

    if *~/.bash_profile* exists, source it,
      else if *~/.bash_login* exists, source it,

else if *~/.profile* exists, source it.

On exit:
if *~/.bash_logout* exists, source it.

Non-login interactive shells:
On startup (subject to the **−norc** and **−rcfile** options):
if *~/.bashrc* exists, source it.

Non-interactive shells:
On startup:
if the environment variable **ENV** is non-null, expand
it and source the file it names, as if the command
if ["$ENV"]; then . $ENV; fi
had been executed, but do not use **PATH** to search
for the pathname. When not started in Posix mode, bash
looks for **BASH_ENV** before **ENV**.

If Bash is invoked as **sh**, it tries to mimic the behavior of **sh** as closely as possible. For a login shell, it attempts to source only */etc/profile* and *~/.profile*, in that order. The **−noprofile** option may still be used to disable this behavior. A shell invoked as **sh** does not attempt to source any other 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, the **ENV** variable is expanded and that file sourced; no other startup files are read.

SEE ALSO

Bash Features, Brian Fox and Chet Ramey
The Gnu Readline Library, Brian Fox and Chet Ramey
The Gnu History Library, Brian Fox and Chet Ramey
A System V Compatible Implementation of 4.2BSD Job Control, David Lennert
Portable Operating System Interface (POSIX) Part 2: Shell and Utilities, IEEE
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
~/.inputrc
Individual *readline* initialization file

AUTHORS

Brian Fox, Free Software Foundation (primary author)
bfox@ai.MIT.Edu

Chet Ramey, Case Western Reserve University
chet@ins.CWRU.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** that you have.

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 welcome to mail that as well! Suggestions and 'philosophical' bug reports may be mailed to *bug-bash@prep.ai.MIT.Edu* or posted 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 behaviour
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@ins.CWRU.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.