CHAPTER 2

GNU Bash Shell Reference

This chapter contains the manual page for the bash shell, produced here for LightStream® 2020 users. The **shell** command in CLI accesses the bash shell so that you can execute LynxOS commands. You can also execute LynxOS commands from the bash shell when you log in as superuser (root).

Numerous commands are built in to the shell (see Shell Builtin Commands). Of particular interest are the commands **cd**, **kill**, and **pwd**. The **help** command displays help information about all the shell builtin commands.

Name

bash - GNU Bourne-Again SHell

Synopsis

```
bash [options] [file]
```

Copyright

Copyright (C) 1989, 1991 by the Free Software Foundation,Inc.

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 faithful 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*.
-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. To be recognized, these options must appear on the command line before the single-character options.

-norc	Do not load the personal initialization file ~/.bashrc if the shell is interactive. This is the default if the shell name is sh.
-noprofile	Do not read either /etc/profile or ~/.bash_profile. By default, bash normally reads these files when it is invoked as a login shell.
-rcfile file	Execute commands from file instead of the standard personal initialization file ~/.bashrc, if the shell is interactive.
-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).
-login	Make bash act as if it had been invoked by login(1).

-nobraceexpansion	Do not perform curly brace expansion as csh does.
-nolineediting	Do not use the GNU readline library to read command lines if interactive.

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.

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:

```
! casedo done elif else esac fi for function if in 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 as follows:

```
[ ! ] 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 optionally terminated by one of ;, &, or <*newline*>.

Of these list operators, && has highest precedence, || has the next highest 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. Commands separated by a ; are executed sequentially; the shell waits for each command to terminate in turn.

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

```
command && command2
```

Here, *command2* is executed if, and only if, *command* returns an exit status of zero. An OR list has the following form:

```
command || command2
```

Here, *command2* is executed if and only if *command* returns a nonzero exit status.

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.
- { *list*; }
  list is simply executed in the current shell environment. This is known as a *group command*.
- 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 in *word* is omitted, the for command executes *list* once for each positional parameter that is set (see Parameters below). The exit status is the exit status of the last command, 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. 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, a word beginning with # causes that word and all remaining characters on that line to be ignored.

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, it is treated as a line continuation (that is, it is effectively ignored), if the backslash is non-quoted.

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. 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 "**$1c$2c...**", 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. 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.
EUID	Expands to the effective user ID of the current user.
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.
OPTARG	The value of the last option argument processed by the getopts builtin command (see Shell Builtin Commands below).
OPTIND	The index of the last option processed by the getopts builtin command (see Shell Builtin Commands below).

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: :/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:/etc:/usr/etc Note that in some circumstances, however, a leading '.' in PATH can be a security hazard.
HOME	The home directory of the current user; the default argument for the cd builtin command.
CDPATH	The search path for the cd builtin 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 (for example, /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 like PS1 and used as the secondary prompt string. The default is "> ".

PS4	The value of this parameter is expanded like PS1 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 “+”.
NO_PROMPT_VARS	If set, the decoded prompt string does not undergo further expansion (see Prompting below).
HISTSIZE	The number of commands to remember in the command history (see History below).
HISTFILE	The name of the file in which command history is saved. (See History below.)
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.
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 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 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.
HOSTTYPE	Automatically set to a string that uniquely describes the type of machine on which bash is executing. The default is system-dependent.
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:~”.
notify	If set, bash reports terminated background jobs immediately, rather than waiting until before printing the next primary prompt.
history_control	If set to a value of *ignorespace*, it means don’t enter lines which begin with a <space> on the history list. If set to a value of *ignoredups*, it means don’t enter lines which match the last entered line. 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) to expand to a null string, rather than themselves.
`histchars`	The two characters which control history expansion and tokenization. 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 character which signifies that the remainder of the line is a comment, when found as the first character of a word.
`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 such as `cd`.
`hostname_completion_file`	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. You can change the file interactively; the next time you want to complete a hostname, `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 in Shell Builtin Commands).
`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.
`no_exit_on_failed_exec`	If this variable exists, the shell does not exit if it cannot execute the file specified in the `exec` command.
`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.
`pushd_silent`	If set, the `pushd` and `popd` builtin commands do not print the current directory stack after successful execution.

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.

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 appended to the beginning of each string contained within the braces, and the postamble is then appended to the end of 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.

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 is substituted. If a ‘-’ follows, the value of OLDPWD is used.

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 PATH, MAILPATH, and CDPATH, and the shell exports the expanded variables.

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. The format for arithmetic expansion is:

```
$[expression]
```

The *expression* is treated as if it were within double quotes, but a double quote inside the braces 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.

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; otherwise each occurrence of an IFS character is 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, bash scans each *word* for the characters *, ?, and [, unless the -f flag has been set. 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, the word is removed if no matches are found. 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 appear anywhere in a *simple command* or may precede or 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.

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 variable noclobber is not consulted, and the file is created regardless of the value of noclobber (see 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.

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.

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*.

Functions

A shell function, defined as described above under Shell Grammar, stores a series of commands for later execution. However, 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 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. The first word of each command 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. 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.

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.

Aliases are not expanded when the shell is not interactive.

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 single process in the job is 25647. 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, causes the process to stop.

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, would refer 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 variable notify is set, bash reports such changes immediately. (See also the -o notify option to the set builtin command under Shell Builtin Commands.)

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 **wait** 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. 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, using 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 under Shell Builtin Commands), then *all* parameter assignments are placed in the environment for a command, not just those that precede the command name.

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.

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 under Shell Builtin Commands.

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

\t	the time
\d	the date
\n	`CRLF`
\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	character code in octal
\\\\	a backslash

After the string is decoded, if the variable NO_PROMPT_VARS is not set, 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. 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. 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 command universal-argument.

The following symbolic character names are recognized:

RUBOUT, DEL, ESC, NEWLINE, SPACE, RETURN, LFD, TAB.

Placing

```
set editing-mode vi
```

into a ~/.inputrc file causes bash to start with a vi-like editing mode. The editing mode may be switched during interactive use by using the -o option to the set builtin command (see under Shell Builtin Commands below).

You can have readline use a single line for display, scrolling the input between the two borders by placing

```
set horizontal-scroll-mode On
```

into a ~/.inputrc file.

The following is a list of the names of the commands and the default key-strokes to get them.

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.
`backward-word`	`M-b`	Move back to the start of this, or the previous, word.
`clear-screen`	`C-l`	Clear the screen leaving the current line at the top of the screen.

Commands for Manipulating the History

`accept-line`	`Newline` `or` `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 history_control variable. If this line was a 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, the first line entered.
`end-of-history`	`M->`	Move to the end of the input history, i.e., the line you are entering.
`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.
`shell-expand-line`	`M-C-e`	Expand the line the way the shell does when it reads it. This performs alias and history expansion. See History below.
`insert-last-argument`	`M-. or M-_`	Insert the last argument to the previous command (the last word on the previous line).
`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 file for editing.

Commands for Changing Text

`delete-char`	`C-d`	Delete the character under the cursor. If the cursor is at the beginning of the line, and 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. A numeric arg says to kill the characters instead of deleting them.
`quoted-insert`	`C-q` or `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`	`M-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. This saves the killed text on the kill-ring (see below).
`backward-kill-line`		Kill backward to the beginning of the line. This is normally unbound, in favor of unix-line-discard, which emulates the behavior of the standard Unix terminal driver.
`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.
`backward-kill-word`	`M-Rubout`	Kill the word behind the cursor.
`unix-line-discard`	`C-u`	Do what C-u used to do in Unix line input. We save the killed text on the kill-ring, though.
`unix-word-rubout`	`C-w`	Do what C-w used to do in Unix line input. The killed text is saved on the kill-ring. This is different than backward-kill-word because the word boundaries differ.
`yank`	`C-y`	Yank the top of the kill ring into the buffer at point.

`yank-pop`	`M-y`	Rotate the kill-ring, and yank the new top. Only works following yank or yank-pop.

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`		Do what C-u does in emacs. 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 $), username (if the text begins with ~), hostname (if the text begins with @), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.
`possible-completions`	`M-?`	List the possible completions of the text before point.
`complete-filename`	`M-/`	Attempt filename completion on the text before point.
`possible-filename-completions`	`C-x /`	List the possible completions of the text before point, treating it as a filename.
`complete-username`	`M-~`	Attempt completion on the text before point, treating it as a username.
`possible-username-completions`	`C-x ~`	List the possible completions of the text before point, treating it as a username.
`complete-variable`	`M-$`	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.

Miscellaneous

`abort`	`C-g`	Abort the current editing command and ring the terminal's bell.
`do-uppercase-version`	`M-a`, `M-b`, ...	Run the command that is bound to the uppercased key.
`prefix-meta`	`ESC`	Metafy the next character typed. This is for people without a meta key. ESC f is equivalent to Meta-f.

`undo`	`C-_`	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 get back to the beginning.
`display-shell-version`	`C-x C-v`	Display version information about the current instance of bash.
`emacs-editing-mode`	`C-e`	When in vi editing mode, this causes a switch to emacs editing mode.
`vi-editing-mode`	`M-C-j` or `M-C-m`	When in emacs editing mode, this causes a switch to vi editing mode.

History

The shell supports a history expansion feature that is similar to the history expansion in csh. This section describes what syntax features are available.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is determining 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 English, or Unix, words surrounded by quotes are considered as one word. Only backslash (\) can quote the history escape character, which is ! by default.

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 <space>, <tab>, <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*.

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).

#	The entire command line typed so far. This means the current command, not the previous command, so it really isn’t a word designator, and doesn’t belong in this section.
`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.

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 suffix.
`t`	Remove all leading pathname components, leaving the tail.
`p`	Print the new command but do not execute it. This takes effect immediately, so it should be the last specifier on the line.

Arithmetic Evaluation

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see Arithmetic Expansion, and see the let builtin command under Shell Builtin Commands). 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
`!`	Logical NOT
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`<= >= < >`	Comparison
`== !=`	Equality and inequality
`=`	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.

Operators are evaluated in order of precedence. Subexpressions 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*
  source *filename*
  Read and execute commands from *filename* in the current shell environment and return the exit status of the last command executed from *filename*. Pathnames in PATH are used to find the directory containing *filename*, if *filename* does not contain a slash. The file searched for in PATH need not be executable. The current directory is searched if no file is found in PATH. The return status is the status of the last command exited within the script (true 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. 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.
- **bind** [**-lvd**] [**-q** *name*]
  bind -f *filename*
  bind *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:

-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*

- **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 command, 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.

- **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** [**-p**] [*command* [*arg* ...]]
  Run *command* with *arg*s 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. An argument of -- disables option checking for the rest of the arguments. If 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.

-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 Expansion) 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.

- **dirs** [**-l**]
  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. The -l option produces a longer listing; the default listing format uses a tilde to denote the home directory.
- **echo** [**-ne**] [*arg* ...]
  Output the *args*, separated by spaces. If **-n** is specified, the trailing newline is suppressed. If the **-e** option is given, interpretation of the following backslash-escaped characters is enabled:

\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**] [*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, *name*s are enabled. For example, to use the test found in PATH instead of the shell builtin version, type enable -n test.
- **eval** [*arg* ...] The *args* 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 *argument*s 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, the shell exits, unless the shell variable no_exit_on_failed_exec exists. If *command* is not specified, any redirections take effect in the current shell.
- **exit** [*n*]

  bye [*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** [-**npf**] [*name*[=*word*]] ... The supplied *name*s are marked for automatic export to the environment of subsequently executed commands. If the **-f** option is given, the *name*s refer to functions. If no *name*s 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 true unless an illegal option is encountered.
- **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, the command is re-executed after the substitution *old*=*new* is performed. 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.

- **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.

- **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 is 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.

- **history** [*n*]
  history -rwan [*filename*]
  With no options, display the command history list with line numbers. Lines listed with 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 (default ~/.bash_history) 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.

- **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.

  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*.

- 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. 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.

- **let** *arg* [*arg* ...]
  Each *arg* is an arithmetic expression to be evaluated (see Arithmetic Expansion). If the last *arg* evaluates to 0, **let** returns 1; 0 is returned otherwise.

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

- **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 variable pushd_silent is unset and the **popd** command is successful, a **dirs** is performed as well.

- **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.

+*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 variable pushd_silent is not set and the **pushd** command is successful, a **dirs** is performed as well.

- **pwd**
  Print the absolute pathname of the current working directory. The path printed contains no symbolic links (but see the description of nolinks under Shell Variables above).
- **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. 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** [**-pf**] [*name* ...]
  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.
- **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.
- **set** [**-aefhknotuvxldCH**] [*arg* ...]

-a	Automatically mark variables which are modified or created for export to the environment of subsequent commands.
-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 by means of **!**.
-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*	The *option* can be one of the following:

allexport	Same as -a.
braceexpand	The shell performs curly brace expansion (see Brace Expansion above). This is on by default.

emacs	Use an emacs-style command line editing interface.
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).
monitor	Same as -m.
noclobber	Same as -C.
noexec	Same as -n.
noglob	Same as -f.
nohash	Same as -d.
notify	The effect is as if the shell command notify= had been executed (see Shell Variables above).
nounset	Same as -u.
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.

-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 by default.
--	If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the args, even if some of them begin with a -.
-	Signal the end of options, cause all remaining args to be assigned to the positional parameters. The -x and -v options are turned off. If there are no args, the positional parameters remain unchanged.

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 args are treated as values for the positional parameters and are assigned, in order, to $1, $2, ... $9. If no options or args 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 .... If *n* is not given, it is assumed to be 1. The exit status is 1 if *n* is greater than $#; 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.
- **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.

-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. If *fd* is omitted, it defaults to 1 (standard output).
-u *file*	True if *file* exists and its set-user-id bit is set.
-w *file*	True if *file* exists and is writeable.
-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 *string*s are equal.
string1 **!=** *string2*	True if the *string*s 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.
- **trap** [*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 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 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 then trap name or number is invalid; otherwise trap returns true.
- **type** [**-all**] [**-type** | **-path**] [*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** [**-SHacdfmstpn** [*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)

The argument **--** 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, and **-p**, which is in units of 512-byte blocks.

- **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.
- **unalias** [*name* ...]
  Remove *name*s from the list of defined aliases. The return value is true unless *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, or LINENO are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless the variable *name* does not exist or is non-unsettable.
- **wait** [*n*]
  Wait for the specified process and report 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 code is zero.

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** flag. PS1 is set and $- includes i if bash is interactive, allowing a way to test this state from a shell script or a startup file.

Login shells:	On login:	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 logout:	If ~/.bash_logout exists, source it.
Non-login interactive shells:	On startup:	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.

See Also

The Gnu Readline Library, Brian Fox *The Gnu History Library,* Brian Fox *A System V Compatible Implementation of 4.2BSD Job Control*, David Lennert *How to wear weird pants for fun and profit*, Brian Fox sh(1), ksh(1), csh(1)

Files

/bin/bash The bash executable /etc/profile The system-wide 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, mail a bug report to bash-maintainers@ai.MIT.Edu. If you have a fix, you are welcome to mail that as well! Suggestions and ‘philosophical’ bug reports may be mailed to bug-bash@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 behavior A short script or ‘recipe’ which exercises the bug

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.