Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'bash.1' command

$ man bash.1

BASH(1) General Commands Manual BASH(1)

NAME

bash - GNU Bourne-Again SHell

SYNOPSIS

bash [options] [command_string | file]

COPYRIGHT

Bash is Copyright (C) 1989-2020 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 incor? porates useful features from the Korn and C shells (ksh and csh).

Bash is intended to be a conformant implementation of the Shell and Utilities portion of the IEEE POSIX specification (IEEE Standard 1003.1). Bash can be configured to be POSIX-conformant by default.

OPTIONS

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

-c If the -c option is present, then commands are read from the first non-option argument command_string. If there are argu? ments after the command_string, the first argument is as? signed to $0 and any remaining arguments are assigned to the

positional parameters. The assignment to $0 sets the name of the shell, which is used in warning and error messages.

-i If the -i option is present, the shell is interactive.

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

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

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

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

[-+]O [shopt_option]

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

-- A -- signals the end of options and disables further option processing. Any arguments after the -- are treated as file? names and arguments. An argument of - is equivalent to --.

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

--debugger

Arrange for the debugger profile to be executed before the shell

starts. Turns on extended debugging mode (see the description of the extdebug option to the shopt builtin below).

--dump-po-strings

Equivalent to -D, but the output is in the GNU gettext po (por? table object) file format.

--dump-strings

Equivalent to -D.

--help Display a usage message on standard output and exit success? fully.

--init-file file

--rcfile file

Execute commands from file instead of the standard personal ini? tialization file ~/.bashrc if the shell is interactive (see IN? VOCATION below).

--login

Equivalent to -l.

--noediting

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

--noprofile

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

--norc Do not read and execute the personal initialization file ~/.bashrc if the shell is interactive. This option is on by de? fault if the shell is invoked as sh.

--posix

Change the behavior of bash where the default operation differs from the POSIX standard to match the standard (posix mode). See SEE ALSO below for a reference to a document that details how posix mode affects bash's behavior.

--restricted

The shell becomes restricted (see RESTRICTED SHELL below).

--rpm-requires

Produce the list of files that are required for the shell script to run. This implies '-n' and is subject to the same limita? tions as compile time error checking checking; Command substitu? tions, Conditional expressions and eval builtin are not parsed so some dependencies may be missed.

--verbose

Equivalent to -v.

--version

Show version information for this instance of bash on the stan? dard output and exit successfully.

ARGUMENTS

If arguments remain after option processing, and neither the -c nor the -s option has been supplied, the first argument is 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 parame? ters are set to the remaining arguments. Bash reads and executes com? mands from this file, then exits. Bash's exit status is the exit sta? tus of the last command executed in the script. If no commands are ex? ecuted, the exit status is 0. An attempt is first made to open the file in the current directory, and, if no file is found, then the shell searches the directories in PATH for the script.

INVOCATION

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

An interactive shell is one started without non-option arguments (un? less -s is specified) and without the -c option whose standard input and error are both connected to terminals (as determined by isatty(3)), or one started with the -i option. PS1 is set and $- includes i if bash is interactive, allowing a shell script or a startup file to test this state.

The following paragraphs describe how bash executes its startup files.
If any of the files exist but cannot be read, bash reports an error.
Tildes are expanded in filenames as described below under Tilde Expan?
sion in the EXPANSION section.
When bash is invoked as an interactive login shell, or as a non-inter?
active shell with the --login option, it first reads and executes com?
mands from the file /etc/profile, if that file exists. After reading
that file, it looks for ~/.bash_profile, ~/.bash_login, and ~/.profile,
in that order, and reads and executes commands from the first one that
exists and is readable. The --noprofile option may be used when the
shell is started to inhibit this behavior.
When an interactive login shell exits, or a non-interactive login shell
executes the exit builtin command, bash reads and executes commands
from the files ~/.bash_logout and /etc/bash.bash_logout, if the files
exists.
When an interactive shell that is not a login shell is started, bash
reads and executes commands from ~/.bashrc, if that file exists. This
may be inhibited by using the --norc option. The --rcfile file option
will force bash to read and execute commands from file instead of
~/.bashrc.
When bash is started non-interactively, to run a shell script, for ex?
ample, it looks for the variable BASH_ENV in the environment, expands
its value if it appears there, and uses the expanded value as the name
of a file to read and execute. Bash behaves as if the following com?
mand were executed:

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

but the value of the PATH variable is not used to search for the file?
name.
If bash is invoked with the name sh, it tries to mimic the startup be?
havior of historical versions of sh as closely as possible, while con?
forming to the POSIX standard as well. When invoked as an interactive
login shell, or a non-interactive shell with the --login option, it
first attempts to read and execute commands from /etc/profile and

~/.profile, in that order. The --noprofile option may be used to in?
hibit this behavior. When invoked as an interactive shell with the
name sh, bash looks for the variable ENV, expands its value if it is
defined, and uses the expanded value as the name of a file to read and
execute. Since a shell invoked as sh does not attempt to read and exe?
cute commands from any other startup files, the --rcfile option has no
effect. A non-interactive shell invoked with the name sh does not at?
tempt to read any other startup files. When invoked as sh, bash enters
posix mode after the startup files are read.

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

Bash attempts to determine when it is being run with its standard input
connected to a network connection, as when executed by the remote shell
daemon, usually rshd, or the secure shell daemon sshd. If bash deter?
mines it is being run in this fashion, it reads and executes commands
from ~/.bashrc, if that file exists and is readable. It will not do
this if invoked as sh. The --norc option may be used to inhibit this
behavior, and the --rcfile option may be used to force another file to
be read, but neither rshd nor sshd generally invoke the shell with
those options or allow them to be specified.

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

DEFINITIONS

The following definitions are used throughout the rest of this docu?
ment.

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 under? scores, and beginning with an alphabetic character or an under? score. Also referred to as an identifier.

metacharacter

A character that, when unquoted, separates words. One of the following:

| & ; ( ) < > space tab newline

control operator

A token that performs a control function. It is one of the fol? lowing symbols:

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

RESERVED WORDS

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

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

SHELL GRAMMAR

Simple Commands

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

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

Pipelines

A pipeline is a sequence of one or more commands separated by one of

the control operators | or |&. The format for a pipeline is:

[time [-p]] [!] command [[|?|&] command2 ...]

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

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

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

When the shell is in posix mode, time may be followed by a newline. In this case, the shell displays the total user and system time consumed by the shell and its children. The TIMEFORMAT variable may be used to specify the format of the time information.

Each command in a pipeline is executed as a separate process (i.e., in a subshell). See COMMAND EXECUTION ENVIRONMENT for a description of a subshell environment. If the lastpipe option is enabled using the

shopt builtin (see the description of shopt below), the last element of a pipeline may be run by the shell process.

Lists

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

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

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

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

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

command1 && command2

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

An OR list has the form

command1 || command2

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

Compound Commands

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

(list) list is executed in a subshell environment (see COMMAND EXECU?
TION ENVIRONMENT below). Variable assignments and builtin com?
mands that affect the shell's environment do not remain in ef?
fect after the command completes. The return status is the exit
status of list.

{ list; }

list is simply executed in the current shell environment. list
must be terminated with a newline or semicolon. This is known
as a group command. The return status is the exit status of
list. Note that unlike the metacharacters (and), { and } are
reserved words and must occur where a reserved word is permitted
to be recognized. Since they do not cause a word break, they
must be separated from list by whitespace or another shell
metacharacter.

((expression))

The expression is evaluated according to the rules described be?
low under ARITHMETIC EVALUATION. If the value of the expression
is non-zero, the return status is 0; otherwise the return status
is 1. This is exactly equivalent to let "expression".

[[expression]]

Return a status of 0 or 1 depending on the evaluation of the
conditional expression expression. Expressions are composed of
the primaries described below under CONDITIONAL EXPRESSIONS.
Word splitting and pathname expansion are not performed on the
words between the [[and]]; tilde expansion, parameter and
variable expansion, arithmetic expansion, command substitution,
process substitution, and quote removal are performed. Condi?
tional operators such as -f must be unquoted to be recognized as
primaries.

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

When the == and != operators are used, the string to the right
of the operator is considered a pattern and matched according to

the rules described below under Pattern Matching, as if the ext? glob shell option were enabled. The = operator is equivalent to ==. If the nocasematch shell option is enabled, the match is performed without regard to the case of alphabetic characters. The return value is 0 if the string matches (==) or does not match (!=) the pattern, and 1 otherwise. Any part of the pat? tern may be quoted to force the quoted portion to be matched as a string.

An additional binary operator, =~, is available, with the same precedence as == and !=. When it is used, the string to the right of the operator is considered a POSIX extended regular ex? pression and matched accordingly (using the POSIX regcomp and regexec interfaces usually described in regex(3)). The return value is 0 if the string matches the pattern, and 1 otherwise. If the regular expression is syntactically incorrect, the condi? tional expression's return value is 2. If the nocasematch shell option is enabled, the match is performed without regard to the case of alphabetic characters. Any part of the pattern may be quoted to force the quoted portion to be matched as a string. Bracket expressions in regular expressions must be treated care? fully, since normal quoting characters lose their meanings be? tween brackets. If the pattern is stored in a shell variable, quoting the variable expansion forces the entire pattern to be matched as a string.

The pattern will match if it matches any part of the string. Anchor the pattern using the ^ and $ regular expression opera? tors to force it to match the entire string. The array variable BASH_REMATCH records which parts of the string matched the pat? tern. The element of BASH_REMATCH with index 0 contains the portion of the string matching the entire regular expression. Substrings matched by parenthesized subexpressions within the regular expression are saved in the remaining BASH_REMATCH in? dices. The element of BASH_REMATCH with index n is the portion

of the string matching the nth parenthesized subexpression.

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

(expression)

Returns the value of expression. This may be used to override the normal precedence of operators.

! expression

True if expression is false.

expression1 && expression2

True if both expression1 and expression2 are true.

expression1 || expression2

True if either expression1 or expression2 is true.

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

for name [[in [word ...]] ;] do list ; done

The list of words following in is expanded, generating a list of items. The variable name is set to each element of this list in turn, and list is executed each time. If the in word is omit? ted, the for command executes list once for each positional pa? rameter that is set (see PARAMETERS below). The return status is the exit status of the last command that executes. If the expansion of the items following in results in an empty list, no commands are executed, and the return status is 0.

for ((expr1 ; expr2 ; expr3)) ; do list ; done

First, the arithmetic expression expr1 is evaluated according to the rules described below under ARITHMETIC EVALUATION. The arithmetic expression expr2 is then evaluated repeatedly until it evaluates to zero. Each time expr2 evaluates to a non-zero value, list is executed and the arithmetic expression expr3 is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in list that is executed, or false if any of the expres?

sions is invalid.

select name [in word] ; do list ; done

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

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

A case command first expands word, and tries to match it against each pattern in turn, using the matching rules described under Pattern Matching below. The word is expanded using tilde expan? sion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution and quote removal. Each pattern examined is expanded using tilde expansion, parame? ter and variable expansion, arithmetic expansion, command sub? stitution, and process substitution. If the nocasematch shell option is enabled, the match is performed without regard to the case of alphabetic characters. When a match is found, the cor? responding list is executed. If the ;; operator is used, no subsequent matches are attempted after the first pattern match. Using ;& in place of ;; causes execution to continue with the list associated with the next set of patterns. Using ;;& in place of ;; causes the shell to test the next pattern list in the statement, if any, and execute any associated list on a suc?

cessful match, continuing the case statement execution as if the pattern list had not matched. The exit status is zero if no pattern matches. Otherwise, it is the exit status of the last command executed in 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 sta? tus of the last command executed, or zero if no condition tested true.

while list-1; do list-2; done

until list-1; do list-2; done

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

Coprocesses

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

The format for a coprocess is:

coproc [NAME] command [redirections]

This creates a coprocess named NAME. If NAME is not supplied, the de? fault name is COPROC. NAME must not be supplied if command is a simple command (see above); otherwise, it is interpreted as the first word of the simple command. When the coprocess is executed, the shell creates an array variable (see Arrays below) named NAME in the context of the

executing shell. The standard output of command is connected via a pipe to a file descriptor in the executing shell, and that file de? scriptor is assigned to NAME[0]. The standard input of command is con? nected via a pipe to a file descriptor in the executing shell, and that file descriptor is assigned to NAME[1]. This pipe is established be? fore any redirections specified by the command (see REDIRECTION below). The file descriptors can be utilized as arguments to shell commands and redirections using standard word expansions. Other than those created to execute command and process substitutions, the file descriptors are not available in subshells. The process ID of the shell spawned to ex? ecute the coprocess is available as the value of the variable NAME_PID. The wait builtin command may be used to wait for the coprocess to ter? minate.

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

Shell Function Definitions

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

fname () compound-command [redirection]

function fname [()] compound-command [redirection]

This defines a function named fname. The reserved word function is optional. If the function reserved word is supplied, the parentheses are optional. The body of the function is the com? pound command compound-command (see Compound Commands above). That command is usually a list of commands between { and }, but may be any command listed under Compound Commands above, with one exception: If the function reserved word is used, but the parentheses are not supplied, the braces are required. com? pound-command is executed whenever fname is specified as the name of a simple command. When in posix mode, fname must be a valid shell name and may not be the name of one of the POSIX

special builtins. In default mode, a function name can be any unquoted shell word that does not contain $. Any redirections (see REDIRECTION below) specified when a function is defined are performed when the function is executed. The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When exe? cuted, the exit status of a function is the exit status of the last command executed in the body. (See FUNCTIONS below.)

COMMENTS

In a non-interactive shell, or an interactive shell in which the inter? active_comments option to the shopt builtin is enabled (see SHELL BUILTIN COMMANDS below), a word beginning with # causes that word and all remaining characters on that line to be ignored. An interactive shell without the interactive_comments option enabled does not allow comments. The interactive_comments option is on by default in interac? tive shells.

QUOTING

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

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

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

There are three quoting mechanisms: the escape character, single quotes, and double quotes.

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

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

Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of $, `, \, and, when history expansion is enabled, !. When the shell is in posix mode, the ! has no special meaning within double quotes, even when history expansion is enabled. The characters $ and ` retain their special meaning within double quotes. The backslash retains its special mean? ing 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. If enabled, history expansion will be performed unless an ! appearing in double quotes is escaped using a backslash. The backslash preceding the ! is not removed.

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

Words of the form $'string' are treated specially. The word expands to string, with backslash-escaped characters replaced as specified by the ANSI C standard. Backslash escape sequences, if present, are decoded as follows:

\a alert (bell)

\b backspace

\e

\E an escape character

\f form feed

\n new line

\r carriage return

\t horizontal tab

\v vertical tab

\\ backslash

\' single quote

\" double quote

\? question mark

\nnn the eight-bit character whose value is the octal value nnn (one to three octal digits)

\xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

\uHHHH the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value HHHH (one to four hex digits)

\UHHHHHHHH the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value HHHHHHHH (one to eight hex digits)

\cx a control-x character

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

A double-quoted string preceded by a dollar sign ($"string") will cause the string to be translated according to the current locale. The get? text infrastructure performs the message catalog lookup and transla? tion, using the LC_MESSAGES and TEXTDOMAIN shell variables. If the current locale is C or POSIX, or if there are no translations avail? able, the dollar sign is ignored. If the string is translated and re? placed, the replacement is double-quoted.

PARAMETERS

A parameter is an entity that stores values. It can be a name, a num? ber, or one of the special characters listed below under Special Param? eters. A variable is a parameter denoted by a name. A variable has a value and zero or more attributes. Attributes are assigned using the declare builtin command (see declare below in SHELL BUILTIN COMMANDS).

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, com? mand substitution, arithmetic expansion, and quote removal (see EXPAN?

SION below). If the variable has its integer attribute set, then value is evaluated as an arithmetic expression even if the $((...)) expansion is not used (see Arithmetic Expansion below). Word splitting is not performed, with the exception of "$@" as explained below under Special Parameters. Pathname expansion is not performed. Assignment state? ments may also appear as arguments to the alias, declare, typeset, ex? port, readonly, and local builtin commands (declaration commands). When in posix mode, these builtins may appear in a command after one or more instances of the command builtin and retain these assignment statement properties.

In the context where an assignment statement is assigning a value to a shell variable or array index, the += operator can be used to append to or add to the variable's previous value. This includes arguments to builtin commands such as declare that accept assignment statements (declaration commands). When += is applied to a variable for which the integer attribute has been set, value is evaluated as an arithmetic ex? pression and added to the variable's current value, which is also eval? uated. When += is applied to an array variable using compound assign? ment (see Arrays below), the variable's value is not unset (as it is when using =), and new values are appended to the array beginning at one greater than the array's maximum index (for indexed arrays) or added as additional key-value pairs in an associative array. When ap? plied to a string-valued variable, value is expanded and appended to the variable's value.

A variable can be assigned the nameref attribute using the -n option to the declare or local builtin commands (see the descriptions of declare and local below) to create a nameref, or a reference to another vari? able. This allows variables to be manipulated indirectly. Whenever the nameref variable is referenced, assigned to, unset, or has its at? tributes modified (other than using or changing the nameref attribute itself), the operation is actually performed on the variable specified by the nameref variable's value. A nameref is commonly used within shell functions to refer to a variable whose name is passed as an argu?

ment to the function. For instance, if a variable name is passed to a shell function as its first argument, running

declare -n ref=$1

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

Positional Parameters

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

Special Parameters

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

* Expands to the positional parameters, starting from one. When the expansion is not within double quotes, each positional pa? rameter expands to a separate word. In contexts where it is performed, those words are subject to further word splitting and pathname expansion. When the expansion occurs within double

quotes, it expands to a single word with the value of each pa? rameter 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 unset, the parameters are separated by spaces. If IFS is null, the parameters are joined without intervening separators.

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

Expands to the number of positional parameters in decimal.

? Expands to the exit status of the most recently executed fore? ground pipeline.

- Expands to the current option flags as specified upon invoca? tion, by the set builtin command, or those set by the shell it? self (such as the -i option).

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

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

0 Expands to the name of the shell or shell script. This is set

at shell initialization. If bash is invoked with a file of com?
mands, $0 is set to the name of that file. If bash is started
with the -c option, then $0 is set to the first argument after
the string to be executed, if one is present. Otherwise, it is
set to the filename used to invoke bash, as given by argument
zero.

Shell Variables

The following variables are set by the shell:

_ At shell startup, set to the pathname used to invoke the shell
or shell script being executed as passed in the environment or
argument list. Subsequently, expands to the last argument to
the previous simple command executed in the foreground, after
expansion. Also set to the full pathname used to invoke each
command executed and placed in the environment exported to that
command. When checking mail, this parameter holds the name of
the mail file currently being checked.

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

BASHOPTS

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

BASHPID

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

BASH_ALIASES

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

BASH_ARGC

An array variable whose values are the number of parameters in each frame of the current bash execution call stack. The number of parameters to the current subroutine (shell function or script executed with . or source) is at the top of the stack. When a subroutine is executed, the number of parameters passed is pushed onto BASH_ARGC. The shell sets BASH_ARGC only when in extended debugging mode (see the description of the extdebug op? tion to the shopt builtin below). Setting extdebug after the shell has started to execute a script, or referencing this vari? able when extdebug is not set, may result in inconsistent val? ues.

BASH_ARGV

An array variable containing all of the parameters in the cur? rent bash execution call stack. The final parameter of the last subroutine call is at the top of the stack; the first parameter of the initial call is at the bottom. When a subroutine is exe? cuted, the parameters supplied are pushed onto BASH_ARGV. The shell sets BASH_ARGV only when in extended debugging mode (see the description of the extdebug option to the shopt builtin be? low). Setting extdebug after the shell has started to execute a script, or referencing this variable when extdebug is not set, may result in inconsistent values.

BASH_ARGV0

When referenced, this variable expands to the name of the shell or shell script (identical to $0; see the description of special parameter 0 above). Assignment to BASH_ARGV0 causes the value

assigned to also be assigned to $0. If BASH_ARGV0 is unset, it loses its special properties, even if it is subsequently reset.

BASH_CMDS

An associative array variable whose members correspond to the internal hash table of commands as maintained by the hash builtin. Elements added to this array appear in the hash table; however, unsetting array elements currently does not cause com? mand names to be removed from the hash table. If BASH_CMDS is unset, it loses its special properties, even if it is subse? quently reset.

BASH_COMMAND

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

BASH_EXECUTION_STRING

The command argument to the -c invocation option.

BASH_LINENO

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

BASH_LOADABLES_PATH

A colon-separated list of directories in which the shell looks for dynamically loadable builtins specified by the enable com? mand.

BASH_REMATCH

An array variable whose members are assigned by the =~ binary operator to the [[conditional command. The element with index 0 is the portion of the string matching the entire regular ex?

pression. The element with index n is the portion of the string matching the nth parenthesized subexpression.

BASH_SOURCE

An array variable whose members are the source filenames where the corresponding shell function names in the FUNCNAME array variable are defined. The shell function ${FUNCNAME[$i]} is de? fined in the file ${BASH_SOURCE[$i]} and called from ${BASH_SOURCE[$i+1]}.

BASH_SUBSHELL

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

BASH_VERSINFO

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

BASH_VERSINFO[0]	The major version number (the release).
BASH_VERSINFO[1]	The minor version number (the version).
BASH_VERSINFO[2]	The patch level.
BASH_VERSINFO[3]	The build version.
BASH_VERSINFO[4]	The release status (e.g., beta1).
BASH_VERSINFO[5]	The value of MACHTYPE.

BASH_VERSION

Expands to a string describing the version of this instance of bash.

COMP_CWORD

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

COMP_KEY

The key (or final key of a key sequence) used to invoke the cur?

rent completion function.

COMP_LINE

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

COMP_POINT

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

COMP_TYPE

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

COMP_WORDBREAKS

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

COMP_WORDS

An array variable (see Arrays below) consisting of the individ? ual words in the current command line. The line is split into words as readline would split it, using COMP_WORDBREAKS as de? scribed above. This variable is available only in shell func? tions invoked by the programmable completion facilities (see Programmable Completion below).

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

DIRSTACK

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

EPOCHREALTIME

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

EPOCHSECONDS

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

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

FUNCNAME

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

it loses its special properties, even if it is subsequently re?
set.

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

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

HISTCMD

The history number, or index in the history list, of the current command. Assignments to HISTCMD are ignored. If HISTCMD is un? set, it loses its special properties, even if it is subsequently reset.

HOSTNAME

Automatically set to the name of the current host.

HOSTTYPE

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

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

MACHTYPE

Automatically set to a string that fully describes the system type on which bash is executing, in the standard GNU cpu-com?

pany-system format. The default is system-dependent.

MAPFILE

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

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

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

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

OSTYPE Automatically set to a string that describes the operating sys? tem on which bash is executing. The default is system-depen? dent.

PIPESTATUS

An array variable (see Arrays below) containing a list of exit status values from the processes in the most-recently-executed foreground pipeline (which may contain only a single command).

PPID The process ID of the shell's parent. This variable is read? only.

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

RANDOM Each time this parameter is referenced, it expands to a random integer between 0 and 32767. Assigning a value to RANDOM ini? tializes (seeds) the sequence of random numbers. If RANDOM is unset, it loses its special properties, even if it is subse? quently reset.

READLINE_LINE

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

READLINE_MARK

The position of the mark (saved insertion point) in the readline line buffer, for use with "bind -x" (see SHELL BUILTIN COMMANDS below). The characters between the insertion point and the mark are often called the region.

READLINE_POINT

The position of the insertion point in the readline line buffer, for use with "bind -x" (see SHELL BUILTIN COMMANDS below).

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

SECONDS

Each time this parameter is referenced, 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. The number of seconds at shell invocation and the current time is always determined by querying the system clock. If SECONDS is unset, it loses its special properties, even if it is subse? quently reset.

SHELLOPTS

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

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

SRANDOM

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

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

The following variables are used by the shell. In some cases, bash as?

signs a default value to a variable; these cases are noted below.

BASH_COMPAT

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

BASH_ENV

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

BASH_XTRACEFD

If set to an integer corresponding to a valid file descriptor, bash will write the trace output generated when set -x is en? abled to that file descriptor. The file descriptor is closed when BASH_XTRACEFD is unset or assigned a new value. Unsetting BASH_XTRACEFD or assigning it the empty string causes the trace output to be sent to the standard error. Note that setting BASH_XTRACEFD to 2 (the standard error file descriptor) and then unsetting it will result in the standard error being closed.

CDPATH The search path for the cd command. This is a colon-separated

list of directories in which the shell looks for destination di?
rectories specified by the cd command. A sample value is
".:~:/usr".

CHILD_MAX

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

COLUMNS

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

COMPREPLY

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

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

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

EXECIGNORE

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

executable bit set, but are not executable files. The pattern matching honors the setting of the extglob shell option.

FCEDIT The default editor for the fc builtin command.

FIGNORE

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

FUNCNEST

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

GLOBIGNORE

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

HISTCONTROL

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

HISTFILE

The name of the file in which command history is saved (see HIS? TORY below). The default value is ~/.bash_history. If unset, the command history is not saved when a shell exits.

HISTFILESIZE

The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is trun? cated, if necessary, to contain no more than that number of lines by removing the oldest entries. The history file is also truncated to this size after writing it when a shell exits. If the value is 0, the history file is truncated to zero size. Non-numeric values and numeric values less than zero inhibit truncation. The shell sets the default value to the value of HISTSIZE after reading any startup files.

HISTIGNORE

A colon-separated list of patterns used to decide which command lines should be saved on the history list. Each pattern is an? chored at the beginning of the line and must match the complete line (no implicit `*' is appended). Each pattern is tested against the line after the checks specified by HISTCONTROL are applied. In addition to the normal shell pattern matching char? acters, `&' matches the previous history line. `&' may be es? caped using a backslash; the backslash is removed before at? tempting a match. The second and subsequent lines of a multi- line compound command are not tested, and are added to the his? tory regardless of the value of HISTIGNORE. The pattern match? ing honors the setting of the extglob shell option.

HISTSIZE

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

HISTTIMEFORMAT

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

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

HOSTFILE

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

IFS The Internal Field Separator that is used for word splitting af? ter expansion and to split lines into words with the read builtin command. The default value is ``<space><tab><new? line>".

IGNOREEOF

Controls the action of an interactive shell on receipt of an EOF character as the sole input. If set, the value is the number of consecutive EOF characters which must be typed as the first characters on an input line before bash exits. If the variable 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.

INPUTRC

The filename for the readline startup file, overriding the de? fault of ~/.inputrc (see READLINE below).

INSIDE_EMACS

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

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

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

LC_COLLATE

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

LC_CTYPE

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

LC_MESSAGES

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

LC_NUMERIC

This variable determines the locale category used for number formatting.

LC_TIME

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

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

receipt of a SIGWINCH.

MAIL If this parameter is set to a file or directory name and the MAILPATH variable is not set, bash informs the user of the ar? rival of mail in the specified file or Maildir-format directory.

MAILCHECK

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

MAILPATH

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

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

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

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

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

``/usr/local/bin:/usr/lo?

cal/sbin:/usr/bin:/usr/sbin:/bin:/sbin''.

POSIXLY_CORRECT

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

PROMPT_COMMAND

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

PROMPT_DIRTRIM

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

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

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

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

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

PS4 The value of this parameter is expanded as with PS1 and the value is printed before each command bash displays during an ex? ecution trace. The first character of the expanded value of PS4 is replicated multiple times, as necessary, to indicate multiple

levels of indirection. The default is ``+ ''.

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

TIMEFORMAT

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

%% A literal %.

%[p][l]R The elapsed time in seconds.

%[p][l]U The number of CPU seconds spent in user mode.

%[p][l]S The number of CPU seconds spent in system mode.

%P The CPU percentage, computed as (%U + %S) / %R.

The optional p is a digit specifying the precision, the number of fractional digits after a decimal point. A value of 0 causes no decimal point or fraction to be output. At most three places after the decimal point may be specified; values of p greater than 3 are changed to 3. If p is not specified, the value 3 is used.

The optional l specifies a longer format, including minutes, of the form MMmSS.FFs. The value of p determines whether or not the fraction is included.

If this variable is not set, bash acts as if it had the value $'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'. If the value is null, no timing information is displayed. A trailing newline is added when the format string is displayed.

TMOUT If set to a value greater than zero, TMOUT is treated as the de? fault timeout for the read builtin. The select command termi? nates if input does not arrive after TMOUT seconds when input is coming from a terminal. In an interactive shell, the value is

interpreted as the number of seconds to wait for a line of input after issuing the primary prompt. Bash terminates after waiting for that number of seconds if a complete line of input does not arrive.

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

auto_resume

This variable controls how the shell interacts with the user and job control. If this variable is set, single word simple com? mands without redirections are treated as candidates for resump? tion of an existing stopped job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, the job most recently accessed is selected. The name of a stopped job, in this context, is the command line used to start it. If set to the value exact, the string supplied must match the name of a stopped job exactly; if set to substring, the string supplied needs to match a substring of the name of a stopped job. The substring value provides functionality analo? gous to the %? job identifier (see JOB CONTROL below). If set to any other value, the supplied string must be a prefix of a stopped job's name; this provides functionality analogous to the %string job identifier.

histchars

The two or three characters which control history expansion and tokenization (see HISTORY EXPANSION below). The first character is the history expansion character, the character which signals the start of a history expansion, normally `!'. The second character is the quick substitution character, which is used as shorthand for re-running the previous command entered, substi? tuting one string for another in the command. The default is `^'. The optional third character is the character which indi? cates that the remainder of the line is a comment when found as the first character of a word, normally `#'. The history com?

ment character causes history substitution to be skipped for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

Arrays

Bash provides one-dimensional indexed and associative array variables. Any variable may be used as an indexed array; the declare builtin will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned con? tiguously. Indexed arrays are referenced using integers (including arithmetic expressions) and are zero-based; associative arrays are ref? erenced using arbitrary strings. Unless otherwise noted, indexed array indices must be non-negative integers.

An indexed array is created automatically if any variable is assigned to using the syntax name[subscript]=value. The subscript is treated as an arithmetic expression that must evaluate to a number. To explicitly declare an indexed array, use declare -a name (see SHELL BUILTIN COM? MANDS below). declare -a name[subscript] is also accepted; the sub? script is ignored.

Associative arrays are created using declare -A name.

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

Arrays are assigned to using compound assignments of the form name=(value1 ... valuen), where each value may be of the form [sub? script]=string. Indexed array assignments do not require anything but string. Each value in the list is expanded using all the shell expan? sions described below under EXPANSION. When assigning to indexed ar? rays, if the optional brackets and subscript are supplied, that index is assigned to; otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero.

When assigning to an associative array, the words in a compound assign? ment may be either assignment statements, for which the subscript is required, or a list of words that is interpreted as a sequence of al? ternating keys and values: name=(key1 value1 key2 value2 ...). These

are treated identically to name=([key1]=value1 [key2]=value2 ...).
The first word in the list determines how the remaining words are in?
terpreted; all assignments in a list must be of the same type. When
using key/value pairs, the keys may not be missing or empty; a final
missing value is treated like the empty string.
This syntax is also accepted by the declare builtin. Individual array
elements may be assigned to using the name[subscript]=value syntax in?
troduced above. When assigning to an indexed array, if name is sub?
scripted by a negative number, that number is interpreted as relative
to one greater than the maximum index of name, so negative indices
count back from the end of the array, and an index of -1 references the
last element.
Any element of an array may be referenced using ${name[subscript]}.
The braces are required to avoid conflicts with pathname expansion. If
subscript is @ or *, the word expands to all members of name. These
subscripts differ only when the word appears within double quotes. If
the word is double-quoted, ${name[*]} expands to a single word with the
value of each array member separated by the first character of the IFS
special variable, and ${name[@]} expands each element of name to a sep?
arate word. When there are no array members, ${name[@]} expands to
nothing. If the double-quoted expansion occurs within a word, the ex?
pansion of the first parameter is joined with the beginning part of the
original word, and the expansion of the last parameter is joined with
the last part of the original word. This is analogous to the expansion
of the special parameters * and @ (see Special Parameters above).
${#name[subscript]} expands to the length of ${name[subscript]}. If
subscript is * or @, the expansion is the number of elements in the ar?
ray. If the subscript used to reference an element of an indexed array
evaluates to a number less than zero, it is interpreted as relative to
one greater than the maximum index of the array, so negative indices
count back from the end of the array, and an index of -1 references the
last element.
Referencing an array variable without a subscript is equivalent to ref?

erencing the array with a subscript of 0. Any reference to a variable using a valid subscript is legal, and bash will create an array if nec? essary.

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

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

The unset builtin is used to destroy arrays. unset name[subscript] de? stroys the array element at index subscript, for both indexed and asso? ciative arrays. Negative subscripts to indexed arrays are interpreted as described above. Unsetting the last element of an array variable does not unset the variable. unset name, where name is an array, or unset name[subscript], where subscript is * or @, removes the entire array.

When using a variable name with a subscript as an argument to a com? mand, such as with unset, without using the word expansion syntax de? scribed above, the argument is subject to pathname expansion. If path? name expansion is not desired, the argument should be quoted.

The declare, local, and readonly builtins each accept a -a option to specify an indexed array and a -A option to specify an associative ar? ray. If both options are supplied, -A takes precedence. The read builtin accepts a -a option to assign a list of words read from the standard input to an array. The set and declare builtins display array values in a way that allows them to be reused as assignments.

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 substitu? tion, arithmetic expansion, word splitting, and pathname expansion.

The order of expansions is: brace expansion; tilde expansion, parameter and variable expansion, arithmetic expansion, and command substitution

(done in a left-to-right fashion); word splitting; and pathname expan? sion.

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

After these expansions are performed, quote characters present in the original word are removed unless they have been quoted themselves (quote removal).

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

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be gener? ated. This mechanism is similar to pathname expansion, but the file? names generated need not exist. Patterns to be brace expanded take the form of an optional preamble, followed by either a series of comma-sep? arated strings or a sequence expression between a pair of braces, fol? lowed by an optional postscript. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example, a{d,c,b}e expands into `ade ace abe'.

A sequence expression takes the form {x..y[..incr]}, where x and y are either integers or single characters, and incr, an optional increment, is an integer. When integers are supplied, the expression expands to each number between x and y, inclusive. Supplied integers may be pre? fixed with 0 to force each term to have the same width. When either x or y begins with a zero, the shell attempts to force all generated terms to contain the same number of digits, zero-padding where neces?

sary. When characters are supplied, the expression expands to each character lexicographically between x and y, inclusive, using the de? fault C locale. Note that both x and y must be of the same type. When the increment is supplied, it is used as the difference between each term. The default increment is 1 or -1 as appropriate.

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

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma or a valid sequence ex? pression. Any incorrectly formed brace expansion is left unchanged. A { or , may be quoted with a backslash to prevent its being considered part of a brace expression. To avoid conflicts with parameter expan? sion, the string ${ is not considered eligible for brace expansion, and inhibits brace expansion until the closing }.

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

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

or

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

Brace expansion introduces a slight incompatibility with historical versions of sh. sh does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. Bash removes braces from words as a consequence of brace expansion. For example, a word entered to sh as file{1,2} appears identically in the output. The same word is output as file1 file2 after expansion by bash. If strict compatibility with sh is desired, start bash with the +B option or disable brace expansion with the +B option to the set com? mand (see SHELL BUILTIN COMMANDS below).

Tilde Expansion

If a word begins with an unquoted tilde character (`~'), all of the characters preceding the first unquoted slash (or all characters, if

there is no unquoted slash) are considered a tilde-prefix. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name. If this login name is the null string, the tilde is replaced with the value of the shell parameter HOME. If HOME is unset, the home direc? tory of the user executing the shell is substituted instead. Other? wise, the tilde-prefix is replaced with the home directory associated with the specified login name.

If the tilde-prefix is a `~+', the value of the shell variable PWD re? places the tilde-prefix. If the tilde-prefix is a `~-', the value of the shell variable OLDPWD, if it is set, is substituted. If the char? acters following the tilde in the tilde-prefix consist of a number N, optionally prefixed by a `+' or a `-', the tilde-prefix is replaced with the corresponding element from the directory stack, as it would be displayed by the dirs builtin invoked with the tilde-prefix as an argu? ment. If the characters following the tilde in the tilde-prefix con? sist of a number without a leading `+' or `-', `+' is assumed.

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

Each variable assignment is checked for unquoted tilde-prefixes immedi? ately following a : or the first =. In these cases, tilde expansion is also performed. Consequently, one may use filenames with tildes in as? signments to PATH, MAILPATH, and CDPATH, and the shell assigns the ex? panded value.

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

Parameter Expansion

The `$' character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which

could be interpreted as part of the name.

When braces are used, the matching ending brace is the first `}' not escaped by a backslash or within a quoted string, and not within an em? bedded arithmetic expansion, command substitution, or parameter expan? sion.

${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. The parameter is a shell parameter as described above PARAMETERS) or an array reference (Arrays).

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

In each of the cases below, word is subject to tilde expansion, parame? ter expansion, command substitution, and arithmetic expansion.

When not performing substring expansion, using the forms documented be? low (e.g., :-), bash tests for a parameter that is unset or null. Omitting the colon results in a test only for a parameter that is un? set.

${parameter:-word}

Use Default Values. If parameter is unset or null, the expan?

sion 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:offset}

${parameter:offset:length}

Substring Expansion. Expands to up to length characters of the value of parameter starting at the character specified by offset. If parameter is @, an indexed array subscripted by @ or *, or an associative array name, the results differ as described below. If length is omitted, expands to the substring of the value of parameter starting at the character specified by offset and extending to the end of the value. length and offset are arithmetic expressions (see ARITHMETIC EVALUATION below). If offset evaluates to a number less than zero, the value is used as an offset in characters from the end of the value of parameter. If length evaluates to a number less than zero, it is interpreted as an offset in characters from the end of the value of parameter rather than a number of characters, and the expansion is the characters between offset and that result. Note that a negative offset must be separated from the colon by at

least one space to avoid being confused with the :- expansion.

If parameter is @, the result is length positional parameters beginning at offset. A negative offset is taken relative to one greater than the greatest positional parameter, so an offset of -1 evaluates to the last positional parameter. It is an expan? sion error if length evaluates to a number less than zero.

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

Substring expansion applied to an associative array produces un? defined results.

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

${!prefix*}

${!prefix@}

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

${!name[@]}

${!name[*]}

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

${#parameter}

Parameter length. The length in characters of the value of pa?

rameter is substituted. If parameter is * or @, the value sub?
stituted is the number of positional parameters. If parameter
is an array name subscripted by * or @, the value substituted is
the number of elements in the array. If parameter is an indexed
array name subscripted by a negative number, that number is in?
terpreted as relative to one greater than the maximum index of
parameter, so negative indices count back from the end of the
array, and an index of -1 references the last element.

${parameter#word}

${parameter##word}

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

${parameter%word}

${parameter%%word}

Remove matching suffix pattern. The word is expanded to produce
a pattern just as in pathname expansion, and matched against the
expanded value of parameter using the rules described under Pat?
tern Matching below. If the pattern matches a trailing portion
of the expanded value of parameter, then the result of the ex?
pansion is the expanded value of parameter with the shortest
matching pattern (the ``%'' case) or the longest matching pat?
tern (the ``%%'' case) deleted. If parameter is @ or *, the

pattern removal operation is applied to each positional parame?
ter in turn, and the expansion is the resultant list. If param?
eter is an array variable subscripted with @ or *, the pattern
removal operation is applied to each member of the array in
turn, and the expansion is the resultant list.

${parameter/pattern/string}

Pattern substitution. The pattern is expanded to produce a pat?
tern just as in pathname expansion, Parameter is expanded and
the longest match of pattern against its value is replaced with
string. The match is performed using the rules described under
Pattern Matching below. If pattern begins with /, all matches
of pattern are replaced with string. Normally only the first
match is replaced. If pattern begins with #, it must match at
the beginning of the expanded value of parameter. If pattern
begins with %, it must match at the end of the expanded value of
parameter. If string is null, matches of pattern are deleted
and the / following pattern may be omitted. If the nocasematch
shell option is enabled, the match is performed without regard
to the case of alphabetic characters. If parameter is @ or *,
the substitution operation is applied to each positional parame?
ter in turn, and the expansion is the resultant list. If param?
eter is an array variable subscripted with @ or *, the substitu?
tion operation is applied to each member of the array in turn,
and the expansion is the resultant list.

${parameter^pattern}

${parameter^^pattern}

${parameter,pattern}

${parameter,,pattern}

Case modification. This expansion modifies the case of alpha?
betic characters in parameter. The pattern is expanded to pro?
duce a pattern just as in pathname expansion. Each character in
the expanded value of parameter is tested against pattern, and,
if it matches the pattern, its case is converted. The pattern

should not attempt to match more than one character. The ^ op? erator converts lowercase letters matching pattern to uppercase; the , operator converts matching uppercase letters to lowercase. The ^^ and ,, expansions convert each matched character in the expanded value; the ^ and , expansions match and convert only the first character in the expanded value. If pattern is omit? ted, it is treated like a ?, which matches every character. If parameter is @ or *, the case modification operation is applied to each positional parameter in turn, and the expansion is the resultant list. If parameter is an array variable subscripted with @ or *, the case modification operation is applied to each member of the array in turn, and the expansion is the resultant list.

${parameter@operator}

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

U The expansion is a string that is the value of parameter with lowercase alphabetic characters converted to upper? case.

u The expansion is a string that is the value of parameter with the first character converted to uppercase, if it is alphabetic.

L The expansion is a string that is the value of parameter with uppercase alphabetic characters converted to lower? case.

Q The expansion is a string that is the value of parameter quoted in a format that can be reused as input.

E The expansion is a string that is the value of parameter with backslash escape sequences expanded as with the $'...' quoting mechanism.

P The expansion is a string that is the result of expanding

the value of parameter as if it were a prompt string (see PROMPTING below).

A The expansion is a string in the form of an assignment statement or declare command that, if evaluated, will recreate parameter with its attributes and value.

K Produces a possibly-quoted version of the value of param?eter, except that it prints the values of indexed and as?sociative arrays as a sequence of quoted key-value pairs (see Arrays above).

a The expansion is a string consisting of flag values rep?resenting parameter's attributes.

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

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

Command Substitution

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

$(command)

or

`command`

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

When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by $, `, or \. The first backquote not preceded by a backslash terminates the command sub?

stitution. 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 backquoted 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 expan? sion is:

$((expression))

The expression is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All to? kens in the expression undergo parameter and variable expansion, com? mand substitution, and quote removal. The result is treated as the arithmetic expression to be evaluated. Arithmetic expansions may be nested.

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

Process Substitution

Process substitution allows a process's input or output to be referred to using a filename. It takes the form of <(list) or >(list). The process list is run asynchronously, and its input or output appears as a filename. This filename is passed as an argument to the current com? mand as the result of the expansion. If the >(list) form is used, writing to the file will provide input for list. If the <(list) form is used, the file passed as an argument should be read to obtain the output of list. Process substitution is supported on systems that sup? port named pipes (FIFOs) or the /dev/fd method of naming open files.

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

Word Splitting

The shell scans the results of parameter expansion, command substitu? tion, 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 using these characters as field terminators. If IFS is unset, or its value is exactly <space><tab><newline>, the default, then sequences of <space>, <tab>, and <newline> at the beginning and end of the results of the previous expansions are ignored, and any sequence of IFS characters not at the beginning or end serves to delimit words. If IFS has a value other than the default, then sequences of the whitespace characters space, tab, and newline are ignored at the beginning and end of the word, as long as the whitespace character is in the value of IFS (an IFS white? space character). Any character in IFS that is not IFS whitespace, along with any adjacent IFS whitespace characters, delimits a field. A sequence of IFS whitespace characters is also treated as a delimiter. If the value of IFS is null, no word splitting occurs.

Explicit null arguments ("" or '') are retained and passed to commands as empty strings. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. If a parame? ter with no value is expanded within double quotes, a null argument re? sults and is retained and passed to a command as an empty string. When a quoted null argument appears as part of a word whose expansion is non-null, the null argument is removed. That is, the word -d'' becomes -d after word splitting and null argument removal.

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

Pathname Expansion

After word splitting, unless the -f option has been set, bash scans each word for the characters *, ?, and [. If one of these characters appears, and is not quoted, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames matching the pattern (see Pattern Matching below). If no matching filenames are

found, and the shell option nullglob is not enabled, the word is left unchanged. If the nullglob option is set, and no matches are found, the word is removed. If the failglob shell option is set, and no matches are found, an error message is printed and the command is not executed. If the shell option nocaseglob is enabled, the match is per? formed without regard to the case of alphabetic characters. When a pattern is used for pathname expansion, the character ``.'' at the start of a name or immediately following a slash must be matched ex? plicitly, unless the shell option dotglob is set. The filenames ``.'' and ``..'' must always be matched explicitly, even if dotglob is set. In other cases, the ``.'' character is not treated specially. When matching a pathname, the slash character must always be matched explic? itly by a slash in the pattern, but in other matching contexts it can be matched by a special pattern character as described below under Pat? tern Matching. See the description of shopt below under SHELL BUILTIN COMMANDS for a description of the nocaseglob, nullglob, failglob, and dotglob shell options.

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

Pattern Matching

Any character that appears in a pattern, other than the special pattern characters described below, matches itself. The NUL character may not

occur in a pattern. A backslash escapes the following character; the escaping backslash is discarded when matching. The special pattern characters must be quoted if they are to be matched literally.

The special pattern characters have the following meanings:

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

? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by a hyphen denotes a range expres? sion; any character that falls between those two charac? ters, inclusive, using the current locale's collating se? quence and character set, is matched. If the first char? acter following the [is a ! or a ^ then any character not enclosed is matched. The sorting order of characters in range expressions is determined by the current locale and the values of the LC_COLLATE or LC_ALL shell vari? ables, if set. To obtain the traditional interpretation of range expressions, where [a-d] is equivalent to [abcd], set value of the LC_ALL shell variable to C, or enable the globasciiranges shell option. A - may be matched by including it as the first or last character in the set. A] may be matched by including it as the first character in the set.

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

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

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

Within [and], an equivalence class can be specified us? ing the syntax [=c=], which matches all characters with the same collation weight (as defined by the current lo? cale) as the character c.

Within [and], the syntax [.symbol.] matches the collat? ing symbol symbol.

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

?(pattern-list)

Matches zero or one occurrence of the given patterns

*(pattern-list)

Matches zero or more occurrences of the given patterns

+(pattern-list)

Matches one or more occurrences of the given patterns

@(pattern-list)

Matches one of the given patterns

!(pattern-list)

Matches anything except one of the given patterns

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

Quote Removal

After the preceding expansions, all unquoted occurrences of the charac? ters \, ', and " that did not result from one of the above expansions are removed.

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment. The following redirection opera? tors may precede or appear anywhere within a simple command or may fol? low a command. Redirections are processed in the order they appear, from left to right.

Each redirection that may be preceded by a file descriptor number may instead be preceded by a word of the form {varname}. In this case, for each redirection operator except >&- and <&-, the shell will allocate a file descriptor greater than or equal to 10 and assign it to varname.

If >&- or <&- is preceded by {varname}, the value of varname defines the file descriptor to close. If {varname} is supplied, the redirect? ion persists beyond the scope of the command, allowing the shell pro? grammer to manage the file descriptor himself.

In the following descriptions, if the file descriptor number is omit? ted, and the first character of the redirection operator is <, the re? direction refers to the standard input (file descriptor 0). If the first character of the redirection operator is >, the redirection refers to the standard output (file descriptor 1).

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

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 from the standard output before the standard out? put was redirected to dirlist.

Bash handles several filenames specially when they are used in redirec? tions, as described in the following table. If the operating system on which bash is running provides these special files, bash will use them; otherwise it will emulate them internally with the behavior described below.

/dev/fd/fd

If fd is a valid integer, file descriptor fd is dupli? cated.

/dev/stdin

File descriptor 0 is duplicated.

/dev/stdout

File descriptor 1 is duplicated.

/dev/stderr

File descriptor 2 is duplicated.

/dev/tcp/host/port

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

/dev/udp/host/port

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

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

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

Redirecting Input

Redirection of input causes the file whose name results from the expan? sion 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 ex?
pansion 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 >, and the noclobber option to the set
builtin has been enabled, the redirection will fail if the file whose
name results from the expansion of word exists and is a regular file.
If the redirection operator is >|, or the redirection operator is > and
the noclobber option to the set builtin command is not enabled, the re?
direction is attempted even if the file named by word exists.

Appending Redirected Output

Redirection of output in this fashion causes the file whose name re?
sults from the expansion of word to be opened for appending on file de?
scriptor n, or the standard output (file descriptor 1) if n is not
specified. If the file does not exist it is created.

The general format for appending output is:

[n]>>word

Redirecting Standard Output and Standard Error

This construct 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.

There are two formats for redirecting standard output and standard er?
ror:

&>word

and

>&word

Of the two forms, the first is preferred. This is semantically equiva?

lent to

>word 2>&1

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

Appending Standard Output and Standard Error

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

The format for appending standard output and standard error is:

&>>word

This is semantically equivalent to

>>word 2>&1

(see Duplicating File Descriptors below).

Here Documents

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

The format of here-documents is:

[n]<<[-]word

here-document

delimiter

No parameter and variable expansion, command substitution, arithmetic expansion, or pathname expansion is performed on word. If any part of word is quoted, the delimiter is the result of quote removal on word, and the lines in the here-document are not expanded. If word is un? quoted, all lines of the here-document are subjected to parameter ex? pansion, command substitution, and arithmetic expansion, the character sequence \<newline> is ignored, and \ must be used to quote the charac? ters \, $, and `.

If the redirection operator is <<-, then all leading tab characters are

stripped from input lines and the line containing delimiter. This al? lows here-documents within shell scripts to be indented in a natural fashion.

Here Strings

A variant of here documents, the format is:

[n]<<<word

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

Duplicating File Descriptors

The redirection operator

[n]<&word

is used to duplicate input file descriptors. If word expands to one or more digits, the file descriptor denoted by n is made to be a copy of that file descriptor. If the digits in word do not specify a file de? scriptor open for input, a redirection error occurs. 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. If the digits in word do not specify a file descriptor open for output, a re? direction error occurs. If word evaluates to -, file descriptor n is closed. As a special case, if n is omitted, and word does not expand to one or more digits or -, the standard output and standard error are redirected as described previously.

Moving File Descriptors

The redirection operator

[n]<&digit-

moves the file descriptor digit to file descriptor n, or the standard

input (file descriptor 0) if n is not specified. digit is closed after being duplicated to n.

Similarly, the redirection operator

[n]>&digit-

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

Opening File Descriptors for Reading and Writing

The redirection operator

[n]<>word

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

ALIASES

Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the alias and unalias builtin commands (see SHELL BUILTIN COMMANDS below). The first word of each simple command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The characters /, $, `, and = and any of the shell metacharacters or quoting characters listed above may not appear in an alias name. The replacement text may contain any valid shell input, including shell metacharacters. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias ls to ls -F, for instance, and bash does not try to recursively expand the replacement text. If the last character of the alias value is a blank, 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. If arguments are needed, a shell function should be used (see FUNCTIONS below).

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

The rules concerning the definition and use of aliases are somewhat confusing. Bash always reads at least one complete line of input, and all lines that make up a compound command, before executing any of the commands on that line or the compound command. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. 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 a function definition is read, not when the function is executed, because a function definition is itself a command. As a consequence, aliases defined in a function are not available until af? ter that function is executed. To be safe, always put alias defini? tions on a separate line, and do not use alias in compound commands.

For almost every purpose, aliases are superseded by shell functions.

FUNCTIONS

A shell function, defined as described above under SHELL GRAMMAR, stores a series of commands for later execution. When the name of a shell function is used as a simple command name, the list of commands associated with that function name is executed. Functions are executed in the context of the current shell; no new process is created to in? terpret 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. Special parameter 0 is unchanged. The first element of the FUNCNAME variable is set to the name of the func? tion while the function is executing.

All other aspects of the shell execution environment are identical be? tween a function and its caller with these exceptions: the DEBUG and RETURN traps (see the description of the trap builtin under SHELL

BUILTIN COMMANDS below) are not inherited unless the function has been given the trace attribute (see the description of the declare builtin below) or the -o functrace shell option has been enabled with the set builtin (in which case all functions inherit the DEBUG and RETURN traps), and the ERR trap is not inherited unless the -o errtrace shell option has been enabled.

Variables local to the function may be declared with the local builtin command. Ordinarily, variables and their values are shared between the function and its caller. If a variable is declared local, the vari? able's visible scope is restricted to that function and its children (including the functions it calls). Local variables "shadow" variables with the same name declared at previous scopes. For instance, a local variable declared in a function hides a global variable of the same name: references and assignments refer to the local variable, leaving the global variable unmodified. When the function returns, the global variable is once again visible.

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

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

The unset builtin also acts using the same dynamic scope: if a variable is local to the current scope, unset will unset it; otherwise the unset will refer to the variable found in any calling scope as described above. If a variable at the current local scope is unset, it will re? main so until it is reset in that scope or until the function returns.

Once the function returns, any instance of the variable at a previous scope will become visible. If the unset acts on a variable at a previ? ous scope, any instance of a variable with that name that had been shadowed will become visible.

The FUNCNEST variable, if set to a numeric value greater than 0, de? fines a maximum function nesting level. Function invocations that ex? ceed the limit cause the entire command to abort.

If the builtin command return is executed in a function, the function completes and execution resumes with the next command after the func? tion call. Any command associated with the RETURN trap is executed be? fore execution resumes. When a function completes, the values of the positional parameters and the special parameter # are restored to the values they had prior to the function's execution.

Function names and definitions may be listed with the -f option to the declare or typeset builtin commands. The -F option to declare or type? set will list the function names only (and optionally the source file and line number, if the extdebug shell option is enabled). Functions may be exported so that subshells automatically have them defined with the -f option to the export builtin. A function definition may be deleted using the -f option to the unset builtin.

Functions may be recursive. The FUNCNEST variable may be used to limit the depth of the function call stack and restrict the number of func? tion invocations. By default, no limit is imposed on the number of re? cursive calls.

ARITHMETIC EVALUATION

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the let and declare builtin commands, the ((com? pound command, and Arithmetic Expansion). Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The fol? lowing list of operators is grouped into levels of equal-precedence op? erators. The levels are listed in order of decreasing precedence.

id++ id--

variable post-increment and post-decrement

- + unary minus and plus

++id --id

variable pre-increment and pre-decrement

! ~ logical and bitwise negation

** exponentiation

* / % multiplication, division, remainder

+ - addition, subtraction

<< >> left and right bitwise shifts

<= >= < >

comparison

== != equality and inequality

& bitwise AND

^ bitwise exclusive OR

| bitwise OR

&& logical AND

|| logical OR

expr?expr:expr

conditional operator

= *= /= %= += -= <<= >>= &= ^= |=

assignment

expr1 , expr2

comma

Shell variables are allowed as operands; parameter expansion is per? formed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the integer attribute using declare -i is assigned a value. A null value evaluates to 0. A shell variable need not have its integer attribute turned on

to be used in an expression.

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

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

CONDITIONAL EXPRESSIONS

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

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

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

When used with [[, the < and > operators sort lexicographically using the current locale. The test command sorts using ASCII ordering.

-a file

True if file exists.

-b file

True if file exists and is a block special file.

-c file

True if file exists and is a character special file.

-d file

True if file exists and is a directory.

-e file

True if file exists.

-f file

True if file exists and is a regular file.

-g file

True if file exists and is set-group-id.

-h file

True if file exists and is a symbolic link.

-k file

True if file exists and its ``sticky'' bit is set.

-p file

True if file exists and is a named pipe (FIFO).

-r file

True if file exists and is readable.

-s file

True if file exists and has a size greater than zero.

-t fd True if file descriptor fd is open and refers to a terminal.

-u file

True if file exists and its set-user-id bit is set.

-w file

True if file exists and is writable.

-x file

True if file exists and is executable.

-G file

True if file exists and is owned by the effective group id.

-L file

True if file exists and is a symbolic link.

-N file

True if file exists and has been modified since it was last read.

-O file

True if file exists and is owned by the effective user id.

-S file

True if file exists and is a socket.

file1 -ef file2

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

file1 -nt file2

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

file1 -ot file2

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

-o optname

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

-v varname

True if the shell variable varname is set (has been assigned a value).

-R varname

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

-z string

True if the length of string is zero.

string

-n string

True if the length of string is non-zero.

string1 == string2

string1 = string2

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

string1 != string2

True if the strings are not equal.

string1 < string2

True if string1 sorts before string2 lexicographically.

string1 > string2

True if string1 sorts after string2 lexicographically.

arg1 OP arg2

OP is one of -eq, -ne, -lt, -le, -gt, or -ge. These arithmetic binary operators return true if arg1 is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to arg2, respectively. Arg1 and arg2 may be positive or negative integers. When used with the [[command, Arg1 and Arg2 are evaluated as arithmetic expressions (see ARITHMETIC EVALUATION above).

SIMPLE COMMAND EXPANSION

When a simple command is executed, the shell performs the following ex? pansions, assignments, and redirections, from left to right, in the following order.

1. The words that the parser has marked as variable assignments (those preceding the command name) and redirections are saved for later processing.
2. The words that are not variable assignments or redirections are expanded. If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.

3. Redirections are performed as described above under REDIRECTION.

4. The text after the = in each variable assignment undergoes tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal before being assigned to the vari? able.

If no command name results, the variable assignments affect the current shell environment. Otherwise, the variables are added to the environ? ment of the executed command and do not affect the current shell envi? ronment. If any of the assignments attempts to assign a value to a readonly variable, an error occurs, and the command exits with a non-zero status.

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

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

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 con? taining an executable file by that name. Bash uses a hash table to re? member the full pathnames of executable files (see hash under SHELL

BUILTIN COMMANDS below). A full search of the directories in PATH is performed only if the command is not found in the hash table. If the search is unsuccessful, the shell searches for a defined shell function named command_not_found_handle. If that function exists, it is invoked in a separate execution environment with the original command and the original command's arguments as its arguments, and the function's exit status becomes the exit status of that subshell. If that function is not defined, the shell prints an error message and returns an exit sta? tus of 127.

If the search is successful, or if the command name contains one or more slashes, the shell executes the named program in a separate execu? tion environment. Argument 0 is set to the name given, and the remain? ing 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 un? der 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 exe? cutable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any.

COMMAND EXECUTION ENVIRONMENT

The shell has an execution environment, which consists of the follow? ing:

? open files inherited by the shell at invocation, as modified by redirections supplied to the exec builtin

? the current working directory as set by cd, pushd, or popd, or

inherited by the shell at invocation

? the file creation mode mask as set by umask or inherited from the shell's parent

? current traps set by trap

? shell parameters that are set by variable assignment or with set or inherited from the shell's parent in the environment

? shell functions defined during execution or inherited from the shell's parent in the environment

? options enabled at invocation (either by default or with com? mand-line arguments) or by set

? options enabled by shopt

? shell aliases defined with alias

? various process IDs, including those of background jobs, the value of $$, and the value of PPID

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

? the shell's open files, plus any modifications and additions specified by redirections to the command

? the current working directory

? the file creation mode mask

? shell variables and functions marked for export, along with variables exported for the command, passed in the environment

? traps caught by the shell are reset to the values inherited from the shell's parent, and traps ignored by the shell are ignored

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

Command substitution, commands grouped with parentheses, and asynchro? nous commands are invoked in a subshell environment that is a duplicate of the shell environment, except that traps caught by the shell are re? set to the values that the shell inherited from its parent at invoca? tion. Builtin commands that are invoked as part of a pipeline are also

executed in a subshell environment. Changes made to the subshell envi?
ronment cannot affect the shell's execution environment.

Subshells spawned to execute command substitutions inherit the value of the -e option from the parent shell. When not in posix mode, bash clears the -e option in such subshells.

If a command is followed by a & and job control is not active, the de? fault standard input for the command is the empty file /dev/null. Oth? erwise, the invoked command inherits the file descriptors of the call? ing shell as modified by redirections.

ENVIRONMENT

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

The shell provides several ways to manipulate the environment. On in? vocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for export to child pro? cesses. Executed commands inherit the environment. The export and de? clare -x commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the envi? ronment is modified, the new value becomes part of the environment, re? placing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modi? fied in the shell, less any pairs removed by the unset command, plus any additions via the export and declare -x commands.

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

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

When bash invokes an external command, the variable _ is set to the full filename of the command and passed to that command in its environ?

ment.

EXIT STATUS

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

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

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

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

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

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

SIGNALS

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

Non-builtin commands run by bash have signal handlers set to the values inherited by the shell from its parent. When job control is not in ef? fect, asynchronous commands ignore SIGINT and SIGQUIT in addition to

these inherited handlers. Commands run as a result of command substi?
tution ignore the keyboard-generated job control signals SIGTTIN, SIGT?
TOU, and SIGTSTP.

The shell exits by default upon receipt of a SIGHUP. Before exiting, an interactive shell resends the SIGHUP to all jobs, running or stopped. Stopped jobs are sent SIGCONT to ensure that they receive the SIGHUP. To prevent the shell from sending the signal to a particular job, it should be removed from the jobs table with the disown builtin (see SHELL BUILTIN COMMANDS below) or marked to not receive SIGHUP us?
ing disown -h.

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

If bash is waiting for a command to complete and receives a signal for which a trap has been set, the trap will not be executed until the com?
mand completes. When bash is waiting for an asynchronous command via the wait builtin, the reception of a signal for which a trap has been set will cause the wait builtin to return immediately with an exit sta?
tus greater than 128, immediately after which the trap is executed.

JOB CONTROL

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

The shell associates a job with each pipeline. It keeps a table of currently executing jobs, which may be listed with the jobs command. When bash starts a job asynchronously (in the background), it prints a line that looks like:

```
[1] 25647
```

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

To facilitate the implementation of the user interface to job control, the operating 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-gen? erated signals. Only foreground processes are allowed to read from or, if the user so specifies with stty tostop, write to the terminal. Background processes which attempt to read from (write to when stty tostop is in effect) the terminal are sent a SIGTTIN (SIGTTOU) signal by the kernel's terminal driver, which, unless caught, suspends the process.

If the operating system on which bash is running supports job control, bash contains facilities to use it. Typing the suspend character (typ? ically ^Z, Control-Z) while a process is running causes that process to be stopped and returns control 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 re? turned to bash. The user 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 charac? ter % introduces a job specification (jobspec). Job number n may be referred to as %n. A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, %ce refers to a stopped job whose command name be? gins with ce. If a prefix matches more than one job, bash reports an error. Using %?ce, on the other hand, refers to any job containing the string ce in its command line. If the substring matches more than one job, bash reports an error. The symbols %% and %+ refer to the shell's

notion of the current job, which is the last job stopped while it was in the foreground or started in the background. The previous job may be referenced using %-. If there is only a single job, %+ and %- can both be used to refer to that job. In output pertaining to jobs (e.g., the output of the jobs command), the current job is always flagged with a +, and the previous job with a -. A single % (with no accompanying job specification) also refers to the current job.

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

The shell learns immediately whenever a job changes state. Normally, bash waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the -b option to the set builtin command is enabled, bash reports such changes immediately. Any trap on SIGCHLD is executed for each child that ex? its.

If an attempt to exit bash is made while jobs are stopped (or, if the checkjobs shell option has been enabled using the shopt builtin, run? ning), the shell prints a warning message, and, if the checkjobs option is enabled, lists the jobs and their statuses. The jobs command may then be used to inspect their status. If a second attempt to exit is made without an intervening command, the shell does not print another warning, and any stopped jobs are terminated.

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

PROMPTING

When executing interactively, bash displays the primary prompt PS1 when it is ready to read a command, and the secondary prompt PS2 when it needs more input to complete a command. Bash displays PS0 after it reads a command but before executing it. Bash displays PS4 as de?

scribed above before tracing each command when the -x option is en? abled. Bash allows these prompt strings to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

\a an ASCII bell character (07)

\d the date in "Weekday Month Date" format (e.g., "Tue May 26")

\D{format}

the format is passed to strftime(3) and the result is in? serted into the prompt string; an empty format results in a locale-specific time representation. The braces are required

\e an ASCII escape character (033)

\h the hostname up to the first `.'

\H the hostname

\j the number of jobs currently managed by the shell

\l the basename of the shell's terminal device name

\n newline

\r carriage return

\s the name of the shell, the basename of $0 (the portion following the final slash)

\t the current time in 24-hour HH:MM:SS format

\T the current time in 12-hour HH:MM:SS format

\@ the current time in 12-hour am/pm format

\A the current time in 24-hour HH:MM format

\u the username of the current user

\v the version of bash (e.g., 2.00)

\V the release of bash, version + patch level (e.g., 2.00.0)

\w the current working directory, with $HOME abbreviated with a tilde (uses the value of the PROMPT_DIRTRIM vari? able)

\W the basename of the current working directory, with $HOME abbreviated with a tilde

\! the history number of this command

\# the command number of this command

\$ if the effective UID is 0, a #, otherwise a $

\nnn the character corresponding to the octal number nnn

\\ a backslash

\[begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt

\] end a sequence of non-printing characters

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

READLINE

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

Readline Notation

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. Simi?

larly, meta keys are denoted by M-key, so M-x means Meta-X. (On key? boards without a meta key, M-x means ESC x, i.e., press the Escape key then the x key. This makes ESC the meta prefix. The combination M-C-x means ESC-Control-x, or press the Escape key then hold the Control key while pressing the x key.)

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

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

Readline Initialization

Readline is customized by putting commands in an initialization file (the inputrc file). The name of this file is taken from the value of the INPUTRC variable. If that variable is unset, the default is ~/.in? putrc. If that file does not exist or cannot be read, the ultimate default is /etc/inputrc. When a program which uses the readline li? brary starts up, the initialization file is read, and the key bindings and variables are set. There are only a few basic constructs allowed in the readline initialization file. Blank lines are ignored. Lines beginning with a # are comments. Lines beginning with a $ indicate conditional constructs. Other lines denote key bindings and variable settings.

The default key-bindings 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 readline command univer?

sal-argument.

The following symbolic character names are recognized: RUBOUT, DEL,

ESC, LFD, NEWLINE, RET, RETURN, SPC, SPACE, and TAB.

In addition to command names, readline allows keys to be bound to a

string that is inserted when the key is pressed (a macro).

Readline Key Bindings

The syntax for controlling key bindings in the inputrc file is simple.

All that is required is the name of the command or the text of a macro

and a key sequence to which it should be bound. The name may be speci?

fied in one of two ways: as a symbolic key name, possibly with Meta- or

Control- prefixes, or as a key sequence.

When using the form keyname:function-name or macro, keyname is the name

of a key spelled out in English. For example:

Control-u: universal-argument

Meta-Rubout: backward-kill-word

Control-o: "> output"

In the above example, C-u is bound to the function universal-argument,

M-DEL is bound to the function backward-kill-word, and C-o is bound to

run the macro expressed on the right hand side (that is, to insert the

text ``> output'' into the line).

In the second form, "keyseq":function-name or macro, keyseq differs

from keyname above in that strings denoting an entire key sequence may

be specified by placing the sequence within double quotes. Some GNU

Emacs style key escapes can be used, as in the following example, but

the symbolic character names are not recognized.

"\C-u": universal-argument

"\C-x\C-r": re-read-init-file

"\e[11~": "Function Key 1"

In this example, C-u is again bound to the function universal-argument.

C-x C-r is bound to the function re-read-init-file, and ESC [1 1 ~ is

bound to insert the text ``Function Key 1''.

The full set of GNU Emacs style escape sequences is

\C- control prefix

\M- meta prefix

\e an escape character

\\ backslash

\" literal "

\' literal '

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

\a alert (bell)

\b backspace

\d delete

\f form feed

\n newline

\r carriage return

\t horizontal tab

\v vertical tab

\nnn the eight-bit character whose value is the octal value nnn (one to three digits)

\xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

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

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

Readline Variables

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

form

set variable-name value

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

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

bell-style (audible)

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

bind-tty-special-chars (On)

If set to On, readline attempts to bind the control characters treated specially by the kernel's terminal driver to their readline equivalents.

blink-matching-paren (Off)

If set to On, readline attempts to briefly move the cursor to an opening parenthesis when a closing parenthesis is inserted.

colored-completion-prefix (Off)

If set to On, when listing completions, readline displays the common prefix of the set of possible completions using a different color. The color definitions are taken from the value of the LS_COLORS environment variable.

colored-stats (Off)

If set to On, readline displays possible completions using different colors to indicate their file type. The color definitions are taken from the value of the LS_COLORS environment variable.

comment-begin (``#'')

The string that is inserted when the readline insert-comment command is executed. This command is bound to M-# in emacs mode

and to # in vi command mode.

completion-display-width (-1)

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

completion-ignore-case (Off)

If set to On, readline performs filename matching and completion in a case-insensitive fashion.

completion-map-case (Off)

If set to On, and completion-ignore-case is enabled, readline treats hyphens (-) and underscores (_) as equivalent when per? forming case-insensitive filename matching and completion.

completion-prefix-display-length (0)

The length in characters of the common prefix of a list of pos? sible completions that is displayed without modification. When set to a value greater than zero, common prefixes longer than this value are replaced with an ellipsis when displaying possi? ble completions.

completion-query-items (100)

This determines when the user is queried about viewing the num? ber of possible completions generated by the possible-comple? tions command. It may be set to any integer value greater than or equal to zero. If the number of possible completions is greater than or equal to the value of this variable, readline will ask whether or not the user wishes to view them; otherwise they are simply listed on the terminal.

convert-meta (On)

If set to On, readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prefixing an escape character (in effect, using escape as the meta prefix). The default is On, but readline will set it to

Off if the locale contains eight-bit characters.

disable-completion (Off)

If set to On, readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to self-insert.

echo-control-characters (On)

When set to On, on operating systems that indicate they support it, readline echoes a character corresponding to a signal gener? ated from the keyboard.

editing-mode (emacs)

Controls whether readline begins with a set of key bindings sim? ilar to Emacs or vi. editing-mode can be set to either emacs or vi.

emacs-mode-string (@)

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

enable-bracketed-paste (On)

When set to On, readline will configure the terminal in a way that will enable it to insert each paste into the editing buffer as a single string of characters, instead of treating each char? acter as if it had been read from the keyboard. This can pre? vent pasted characters from being interpreted as editing com? mands.

enable-keypad (Off)

When set to On, readline will try to enable the application key? pad when it is called. Some systems need this to enable the ar? row keys.

enable-meta-key (On)

When set to On, readline will try to enable any meta modifier key the terminal claims to support when it is called. On many terminals, the meta key is used to send eight-bit characters.

expand-tilde (Off)

If set to On, tilde expansion is performed when readline attempts word completion.

history-preserve-point (Off)

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

history-size (unset)

Set the maximum number of history entries saved in the history list. If set to zero, any existing history entries are deleted and no new entries are saved. If set to a value less than zero, the number of history entries is not limited. By default, the number of history entries is set to the value of the HISTSIZE shell variable. If an attempt is made to set history-size to a non-numeric value, the maximum number of history entries will be set to 500.

horizontal-scroll-mode (Off)

When set to On, makes readline use a single line for display, scrolling the input horizontally on a single screen line when it becomes longer than the screen width rather than wrapping to a new line. This setting is automatically enabled for terminals of height 1.

input-meta (Off)

If set to On, readline will enable eight-bit input (that is, it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The name meta-flag is a synonym for this variable. The default is Off, but readline will set it to On if the locale contains eight-bit characters.

isearch-terminators (``C-[C-J'')

The string of characters that should terminate an incremental search without subsequently executing the character as a com? mand. If this variable has not been given a value, the charac? ters ESC and C-J will terminate an incremental search.

keymap (emacs)

Set the current readline keymap. The set of valid keymap names is emacs, emacs-standard, emacs-meta, emacs-ctlx, vi, vi-com? mand, and vi-insert. vi is equivalent to vi-command; emacs is equivalent to emacs-standard. The default value is emacs; the value of editing-mode also affects the default keymap.

keyseq-timeout (500)

Specifies the duration readline will wait for a character when reading an ambiguous key sequence (one that can form a complete key sequence using the input read so far, or can take additional input to complete a longer key sequence). If no input is re? ceived within the timeout, readline will use the shorter but complete key sequence. The value is specified in milliseconds, so a value of 1000 means that readline will wait one second for additional input. If this variable is set to a value less than or equal to zero, or to a non-numeric value, readline will wait until another key is pressed to decide which key sequence to complete.

mark-directories (On)

If set to On, completed directory names have a slash appended.

mark-modified-lines (Off)

If set to On, history lines that have been modified are dis? played with a preceding asterisk (*).

mark-symlinked-directories (Off)

If set to On, completed names which are symbolic links to direc? tories have a slash appended (subject to the value of mark-di? rectories).

match-hidden-files (On)

This variable, when set to On, causes readline to match files whose names begin with a `.' (hidden files) when performing filename completion. If set to Off, the leading `.' must be supplied by the user in the filename to be completed.

menu-complete-display-prefix (Off)

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

output-meta (Off)

If set to On, readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is Off, but readline will set it to On if the locale contains eight-bit characters.

page-completions (On)

If set to On, readline uses an internal more-like pager to dis? play a screenful of possible completions at a time.

print-completions-horizontally (Off)

If set to On, readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen.

revert-all-at-newline (Off)

If set to On, readline will undo all changes to history lines before returning when accept-line is executed. By default, his? tory lines may be modified and retain individual undo lists across calls to readline.

show-all-if-ambiguous (Off)

This alters the default behavior of the completion functions. If set to On, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell.

show-all-if-unmodified (Off)

This alters the default behavior of the completion functions in a fashion similar to show-all-if-ambiguous. If set to On, words

which have more than one possible completion without any possi?
ble partial completion (the possible completions don't share a
common prefix) cause the matches to be listed immediately in?
stead of ringing the bell.

show-mode-in-prompt (Off)

If set to On, add a string to the beginning of the prompt indi?
cating the editing mode: emacs, vi command, or vi insertion.
The mode strings are user-settable (e.g., emacs-mode-string).

skip-completed-text (Off)

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

vi-cmd-mode-string ((cmd))

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

vi-ins-mode-string ((ins))

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

visible-stats (Off)

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

Readline Conditional Constructs

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

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

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

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

version

The version test may be used to perform comparisons against specific readline versions. The version expands to the current readline version. The set of comparison operators includes =, (and ==), !=, <=, >=, <, and >. The version number supplied on the right side of the op?

erator consists of a major version number, an optional decimal point, and an optional minor version (e.g., 7.1). If the minor version is omitted, it is assumed to be 0. The operator may be separated from the string version and from the version number argument by whitespace.

application

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

$if Bash

Quote the current or previous word

"\C-xq": "\eb\"\ef\""

$endif

variable

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

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

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

$include

This directive takes a single filename as an argument and reads

commands and bindings from that file. For example, the follow?

ing directive would read /etc/inputrc:

$include /etc/inputrc

Searching

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

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, read? line displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. The characters present in the value of the isearch-terminators variable are used to terminate an incremental search. If that variable has not been assigned a value the Escape and Control-J characters will terminate an incremental search. Control-G will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type Control-S or Control-R as appropriate. This will search backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a readline command will terminate the search and execute that command. For instance, a newline will termi? nate the search and accept the line, thereby executing the command from the history list.

Readline remembers the last incremental search string. If two Control-Rs are typed without any intervening characters defining a new search string, any remembered search string is used.

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

Readline Command Names

The following is a list of the names of the commands and the default

key sequences to which they are bound. Command names without an accom?
panying key sequence are unbound by default. In the following descrip?
tions, point refers to the current cursor position, and mark refers to
a cursor position saved by the set-mark command. The text between the
point and mark is referred to as the region.

Commands for Moving

beginning-of-line (C-a)

Move to the start of the current line.

end-of-line (C-e)

Move to the end of the line.

forward-char (C-f)

Move forward a character.

backward-char (C-b)

Move back a character.

forward-word (M-f)

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

backward-word (M-b)

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

shell-forward-word

Move forward to the end of the next word. Words are delimited
by non-quoted shell metacharacters.

shell-backward-word

Move back to the start of the current or previous word. Words
are delimited by non-quoted shell metacharacters.

previous-screen-line

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

next-screen-line

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

clear-display (M-C-l)

Clear the screen and, if possible, the terminal's scrollback buffer, then redraw the current line, leaving the current line at the top of the screen.

clear-screen (C-l)

Clear the screen, then redraw the current line, leaving the cur? rent line at the top of the screen. With an argument, refresh the current line without clearing the screen.

redraw-current-line

Refresh the current line.

Commands for Manipulating the History

accept-line (Newline, Return)

Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the state of the HISTCONTROL variable. If the line is a modified history line, then restore the history line to its original state.

previous-history (C-p)

Fetch the previous command from the history list, moving back in the list.

next-history (C-n)

Fetch the next command from the history list, moving forward in the list.

beginning-of-history (M-<)

Move to the first line in the history.

end-of-history (M->)

Move to the end of the input history, i.e., the line currently being entered.

reverse-search-history (C-r)

Search backward starting at the current line and moving `up' through the history as necessary. This is an incremental search.

forward-search-history (C-s)

Search forward starting at the current line and moving `down' through the history as necessary. This is an incremental search.

non-incremental-reverse-search-history (M-p)

Search backward through the history starting at the current line using a non-incremental search for a string supplied by the user.

non-incremental-forward-search-history (M-n)

Search forward through the history using a non-incremental search for a string supplied by the user.

history-search-forward

Search forward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search.

history-search-backward

Search backward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search.

history-substring-search-backward

Search backward through the history for the string of characters between the start of the current line and the current cursor po? sition (the point). The search string may match anywhere in a history line. This is a non-incremental search.

history-substring-search-forward

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

yank-nth-arg (M-C-y)

Insert the first argument to the previous command (usually the second word on the previous line) at point. With an argument n, insert the nth word from the previous command (the words in the previous command begin with word 0). A negative argument in? serts the nth word from the end of the previous command. Once the argument n is computed, the argument is extracted as if the "!n" history expansion had been specified.

yank-last-arg (M-., M-_)

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

shell-expand-line (M-C-e)

Expand the line as the shell does. This performs alias and his? tory expansion as well as all of the shell word expansions. See HISTORY EXPANSION below for a description of history expansion.

history-expand-line (M-^)

Perform history expansion on the current line. See HISTORY EX? PANSION below for a description of history expansion.

magic-space

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

alias-expand-line

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

history-and-alias-expand-line

Perform history and alias expansion on the current line.

insert-last-argument (M-., M-_)

A synonym for yank-last-arg.

operate-and-get-next (C-o)

Accept the current line for execution and fetch the next line relative to the current line from the history for editing. A numeric argument, if supplied, specifies the history entry to use instead of the current line.

edit-and-execute-command (C-x C-e)

Invoke an editor on the current command line, and execute the result as shell commands. Bash attempts to invoke $VISUAL, $ED? ITOR, and emacs as the editor, in that order.

Commands for Changing Text

end-of-file (usually C-d)

The character indicating end-of-file as set, for example, by ``stty''. If this character is read when there are no charac? ters on the line, and point is at the beginning of the line, Readline interprets it as the end of input and returns EOF.

delete-char (C-d)

Delete the character at point. If this function is bound to the same character as the tty EOF character, as C-d commonly is, see above for the effects.

backward-delete-char (Rubout)

Delete the character behind the cursor. When given a numeric argument, save the deleted text on the kill ring.

forward-backward-delete-char

Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cur? sor is deleted.

quoted-insert (C-q, C-v)

Add the next character typed to the line verbatim. This is how to insert characters like C-q, for example.

tab-insert (C-v TAB)

Insert a tab character.

self-insert (a, b, A, 1, !, ...)

Insert the character typed.

transpose-chars (C-t)

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

transpose-words (M-t)

Drag the word before point past the word after point, moving point over that word as well. If point is at the end of the line, this transposes the last two words on the line.

upcase-word (M-u)

Uppercase the current (or following) word. With a negative ar? gument, uppercase the previous word, but do not move point.

downcase-word (M-l)

Lowercase the current (or following) word. With a negative ar? gument, lowercase the previous word, but do not move point.

capitalize-word (M-c)

Capitalize the current (or following) word. With a negative ar? gument, capitalize the previous word, but do not move point.

overwrite-mode

Toggle overwrite mode. With an explicit positive numeric argu? ment, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only emacs mode; vi mode does overwrite differently. Each call to readline() starts in insert mode. In overwrite mode, charac? ters bound to self-insert replace the text at point rather than pushing the text to the right. Characters bound to back? ward-delete-char replace the character before point with a space. By default, this command is unbound.

kill-line (C-k)

Kill the text from point to the end of the line.

backward-kill-line (C-x Rubout)

Kill backward to the beginning of the line.

unix-line-discard (C-u)

Kill backward from point to the beginning of the line. The killed text is saved on the kill-ring.

kill-whole-line

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

kill-word (M-d)

Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by forward-word.

backward-kill-word (M-Rubout)

Kill the word behind point. Word boundaries are the same as those used by backward-word.

shell-kill-word

Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by shell-forward-word.

shell-backward-kill-word

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

unix-word-rubout (C-w)

Kill the word behind point, using white space as a word bound? ary. The killed text is saved on the kill-ring.

unix-filename-rubout

Kill the word behind point, using white space and the slash character as the word boundaries. The killed text is saved on the kill-ring.

delete-horizontal-space (M-\)

Delete all spaces and tabs around point.

kill-region

Kill the text in the current region.

copy-region-as-kill

Copy the text in the region to the kill buffer.

copy-backward-word

Copy the word before point to the kill buffer. The word bound? aries are the same as backward-word.

copy-forward-word

Copy the word following point to the kill buffer. The word boundaries are the same as forward-word.

yank (C-y)

Yank the top of the kill ring into the buffer at point.

yank-pop (M-y)

Rotate the kill ring, and yank the new top. Only works follow? ing yank or yank-pop.

Numeric Arguments

digit-argument (M-0, M-1, ..., M--)

Add this digit to the argument already accumulating, or start a new argument. M-- starts a negative argument.

universal-argument

This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is fol? lowed by digits, executing universal-argument again ends the nu? meric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is nei? ther a digit nor minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argu? ment count four, a second time makes the argument count sixteen, and so on.

Completing

complete (TAB)

Attempt to perform completion on the text before point. Bash attempts completion treating the text as a variable (if the text begins with $), username (if the text begins with ~), hostname (if the text begins with @), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.

possible-completions (M-?)

List the possible completions of the text before point.

insert-completions (M-*)

Insert all completions of the text before point that would have been generated by possible-completions.

menu-complete

Similar to complete, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of menu-complete steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung (subject to the setting of bell-style) and the original text is restored. An argument of n moves n positions forward in the list of matches; a negative ar? gument may be used to move backward through the list. This com? mand is intended to be bound to TAB, but is unbound by default.

menu-complete-backward

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

delete-char-or-list

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

complete-filename (M-/)

Attempt filename completion on the text before point.

possible-filename-completions (C-x /)

List the possible completions of the text before point, treating it as a filename.

complete-username (M-~)

Attempt completion on the text before point, treating it as a username.

possible-username-completions (C-x ~)

List the possible completions of the text before point, treating it as a username.

complete-variable (M-$)

Attempt completion on the text before point, treating it as a shell variable.

possible-variable-completions (C-x $)

List the possible completions of the text before point, treating it as a shell variable.

complete-hostname (M-@)

Attempt completion on the text before point, treating it as a hostname.

possible-hostname-completions (C-x @)

List the possible completions of the text before point, treating it as a hostname.

complete-command (M-!)

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

possible-command-completions (C-x !)

List the possible completions of the text before point, treating it as a command name.

dynamic-complete-history (M-TAB)

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

dabbrev-expand

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

complete-into-braces (M-{)

Perform filename completion and insert the list of possible com? pletions enclosed within braces so the list is available to the shell (see Brace Expansion above).

Keyboard Macros

start-kbd-macro (C-x ()

Begin saving the characters typed into the current keyboard macro.

end-kbd-macro (C-x))

Stop saving the characters typed into the current keyboard macro and store the definition.

call-last-kbd-macro (C-x e)

Re-execute the last keyboard macro defined, by making the char? acters in the macro appear as if typed at the keyboard.

print-last-kbd-macro ()

Print the last keyboard macro defined in a format suitable for the inputrc file.

Miscellaneous

re-read-init-file (C-x C-r)

Read in the contents of the inputrc file, and incorporate any bindings or variable assignments found there.

abort (C-g)

Abort the current editing command and ring the terminal's bell (subject to the setting of bell-style).

do-lowercase-version (M-A, M-B, M-x, ...)

If the metafied character x is uppercase, run the command that is bound to the corresponding metafied lowercase character. The behavior is undefined if x is already lowercase.

prefix-meta (ESC)

Metafy the next character typed. ESC f is equivalent to Meta-f.

undo (C-_, C-x C-u)

Incremental undo, separately remembered for each line.

revert-line (M-r)

Undo all changes made to this line. This is like executing the undo command enough times to return the line to its initial state.

tilde-expand (M-&)

Perform tilde expansion on the current word.

set-mark (C-@, M-<space>)

Set the mark to the point. If a numeric argument is supplied, the mark is set to that position.

exchange-point-and-mark (C-x C-x)

Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

character-search (C-])

A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occur? rences.

character-search-backward (M-C-])

A character is read and point is moved to the previous occur? rence of that character. A negative count searches for subse? quent occurrences.

skip-csi-sequence

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

This is unbound by default, but usually bound to ESC-[.

insert-comment (M-#)

Without a numeric argument, the value of the readline com?

ment-begin variable is inserted at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of comment-begin, the value is inserted, other? wise the characters in comment-begin are deleted from the begin? ning of the line. In either case, the line is accepted as if a newline had been typed. The default value of comment-begin causes this command to make the current line a shell comment. If a numeric argument causes the comment character to be re? moved, the line will be executed by the shell.

glob-complete-word (M-g)

The word before point is treated as a pattern for pathname ex? pansion, with an asterisk implicitly appended. This pattern is used to generate a list of matching filenames for possible com? pletions.

glob-expand-word (C-x *)

The word before point is treated as a pattern for pathname ex? pansion, and the list of matching filenames is inserted, replac? ing the word. If a numeric argument is supplied, an asterisk is appended before pathname expansion.

glob-list-expansions (C-x g)

The list of expansions that would have been generated by glob-expand-word is displayed, and the line is redrawn. If a numeric argument is supplied, an asterisk is appended before pathname expansion.

dump-functions

Print all of the functions and their key bindings to the read? line output stream. If a numeric argument is supplied, the out? put is formatted in such a way that it can be made part of an inputrc file.

dump-variables

Print all of the settable readline variables and their values to the readline output stream. If a numeric argument is supplied,

the output is formatted in such a way that it can be made part of an inputrc file.

dump-macros

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

display-shell-version (C-x C-v)

Display version information about the current instance of bash.

Programmable Completion

When word completion is attempted for an argument to a command for which a completion specification (a compspec) has been defined using the complete builtin (see SHELL BUILTIN COMMANDS below), the program? mable completion facilities are invoked.

First, the command name is identified. If the command word is the empty string (completion attempted at the beginning of an empty line), any compspec defined with the -E option to complete is used. If a compspec has been defined for that command, the compspec is used to generate the list of possible completions for the word. If the command word is a full pathname, a compspec for the full pathname is searched for first. If no compspec is found for the full pathname, an attempt is made to find a compspec for the portion following the final slash.

If those searches do not result in a compspec, any compspec defined with the -D option to complete is used as the default. If there is no default compspec, bash attempts alias expansion on the command word as a final resort, and attempts to find a compspec for the command word from any successful expansion.

Once a compspec has been found, it is used to generate the list of matching words. If a compspec is not found, the default bash comple? tion as described above under Completing is performed.

First, the actions specified by the compspec are used. Only matches which are prefixed by the word being completed are returned. When the -f or -d option is used for filename or directory name completion, the

shell variable FIGNORE is used to filter the matches.

Any completions specified by a pathname expansion pattern to the -G op?
tion are generated next. The words generated by the pattern need not
match the word being completed. The GLOBIGNORE shell variable is not
used to filter the matches, but the FIGNORE variable is used.

Next, the string specified as the argument to the -W option is consid?
ered. The string is first split using the characters in the IFS spe?
cial variable as delimiters. Shell quoting is honored. Each word is
then expanded using brace expansion, tilde expansion, parameter and
variable expansion, command substitution, and arithmetic expansion, as
described above under EXPANSION. The results are split using the rules
described above under Word Splitting. The results of the expansion are
prefix-matched against the word being completed, and the matching words
become the possible completions.

After these matches have been generated, any shell function or command
specified with the -F and -C options is invoked. When the command or
function is invoked, the COMP_LINE, COMP_POINT, COMP_KEY, and COMP_TYPE
variables are assigned values as described above under Shell Variables.

If a shell function is being invoked, the COMP_WORDS and COMP_CWORD
variables are also set. When the function or command is invoked, the
first argument ($1) is the name of the command whose arguments are be?
ing completed, the second argument ($2) is the word being completed,
and the third argument ($3) is the word preceding the word being com?
pleted on the current command line. No filtering of the generated com?
pletions against the word being completed is performed; the function or
command has complete freedom in generating the matches.

Any function specified with -F is invoked first. The function may use
any of the shell facilities, including the compgen builtin described
below, to generate the matches. It must put the possible completions
in the COMPREPLY array variable, one per array element.

Next, any command specified with the -C option is invoked in an envi?
ronment equivalent to command substitution. It should print a list of
completions, one per line, to the standard output. Backslash may be

used to escape a newline, if necessary.

After all of the possible completions are generated, any filter speci?
fied with the -X option is applied to the list. The filter is a pat?
tern as used for pathname expansion; a & in the pattern is replaced
with the text of the word being completed. A literal & may be escaped
with a backslash; the backslash is removed before attempting a match.
Any completion that matches the pattern will be removed from the list.
A leading ! negates the pattern; in this case any completion not match?
ing the pattern will be removed. If the nocasematch shell option is
enabled, the match is performed without regard to the case of alpha?
betic characters.

Finally, any prefix and suffix specified with the -P and -S options are
added to each member of the completion list, and the result is returned
to the readline completion code as the list of possible completions.

If the previously-applied actions do not generate any matches, and the
-o dirnames option was supplied to complete when the compspec was de?
fined, directory name completion is attempted.

If the -o plusdirs option was supplied to complete when the compspec
was defined, directory name completion is attempted and any matches are
added to the results of the other actions.

By default, if a compspec is found, whatever it generates is returned
to the completion code as the full set of possible completions. The
default bash completions are not attempted, and the readline default of
filename completion is disabled. If the -o bashdefault option was sup?
plied to complete when the compspec was defined, the bash default com?
pletions are attempted if the compspec generates no matches. If the -o
default option was supplied to complete when the compspec was defined,
readline's default completion will be performed if the compspec (and,
if attempted, the default bash completions) generate no matches.

When a compspec indicates that directory name completion is desired,
the programmable completion functions force readline to append a slash
to completed names which are symbolic links to directories, subject to
the value of the mark-directories readline variable, regardless of the

setting of the mark-symlinked-directories readline variable.

There is some support for dynamically modifying completions. This is most useful when used in combination with a default completion speci? fied with complete -D. It's possible for shell functions executed as completion handlers to indicate that completion should be retried by returning an exit status of 124. If a shell function returns 124, and changes the compspec associated with the command on which completion is being attempted (supplied as the first argument when the function is executed), programmable completion restarts from the beginning, with an attempt to find a new compspec for that command. This allows a set of completions to be built dynamically as completion is attempted, rather than being loaded all at once.

For instance, assuming that there is a library of compspecs, each kept in a file corresponding to the name of the command, the following de? fault completion function would load completions dynamically:

```
_completion_loader()
{
    . "/etc/bash_completion.d/$1.sh" >/dev/null 2>&1 && return 124
}
complete -D -F _completion_loader -o bashdefault -o default
```

HISTORY

When the -o history option to the set builtin is enabled, the shell provides access to the command history, the list of commands previously typed. The value of the HISTSIZE variable is used as the number of commands to save in a history list. The text of the last HISTSIZE com? mands (default 500) is saved. The shell stores each command in the history list prior to parameter and variable expansion (see EXPANSION above) but after history expansion is performed, subject to the values of the shell variables HISTIGNORE and HISTCONTROL.

On startup, the history is initialized from the file named by the vari? able HISTFILE (default ~/.bash_history). The file named by the value of HISTFILE is truncated, if necessary, to contain no more than the number of lines specified by the value of HISTFILESIZE. If HISTFILE?

SIZE is unset, or set to null, a non-numeric value, or a numeric value less than zero, the history file is not truncated. When the history file is read, lines beginning with the history comment character fol? lowed immediately by a digit are interpreted as timestamps for the fol? lowing history line. These timestamps are optionally displayed depend? ing on the value of the HISTTIMEFORMAT variable. When a shell with history enabled exits, the last $HISTSIZE lines are copied from the history list to $HISTFILE. If the histappend shell option is enabled (see the description of shopt under SHELL BUILTIN COMMANDS below), the lines are appended to the history file, otherwise the history file is overwritten. If HISTFILE is unset, or if the history file is un? writable, the history is not saved. If the HISTTIMEFORMAT variable is set, time stamps are written to the history file, marked with the his? tory comment character, so they may be preserved across shell sessions. This uses the history comment character to distinguish timestamps from other history lines. After saving the history, the history file is truncated to contain no more than HISTFILESIZE lines. If HISTFILESIZE is unset, or set to null, a non-numeric value, or a numeric value less than zero, the history file is not truncated.

The builtin command fc (see SHELL BUILTIN COMMANDS below) may be used to list or edit and re-execute a portion of the history list. The his? tory builtin may be used to display or modify the history list and ma? nipulate the history file. When using command-line editing, search commands are available in each editing mode that provide access to the history list.

The shell allows control over which commands are saved on the history list. The HISTCONTROL and HISTIGNORE variables may be set to cause the shell to save only a subset of the commands entered. The cmdhist shell option, if enabled, causes the shell to attempt to save each line of a multi-line command in the same history entry, adding semicolons where necessary to preserve syntactic correctness. The lithist shell option causes the shell to save the command with embedded newlines instead of semicolons. See the description of the shopt builtin below under SHELL

BUILTIN COMMANDS for information on setting and unsetting shell op?
tions.

HISTORY EXPANSION

The shell supports a history expansion feature that is similar to the history expansion in csh. This section describes what syntax features are available. This feature is enabled by default for interactive shells, and can be disabled using the +H option to the set builtin com? mand (see SHELL BUILTIN COMMANDS below). Non-interactive shells do not perform history expansion by default.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words, and is performed on each line individually without taking quoting on previous lines into ac? count. It takes place in two parts. The first is to determine which line from the history list to use during substitution. The second is to select portions of that line for inclusion into the current one.

The line selected from the history is the event, and the portions of that line that are acted upon are words. Various modifiers are avail? able to manipulate the selected words. The line is broken into words in the same fashion as when reading input, so that several metacharac? ter-separated words surrounded by quotes are considered one word. His? tory expansions are introduced by the appearance of the history expan? sion character, which is ! by default. Only backslash (\) and single quotes can quote the history expansion character, but the history ex? pansion character is also treated as quoted if it immediately precedes the closing double quote in a double-quoted string.

Several characters inhibit history expansion if found immediately fol? lowing the history expansion character, even if it is unquoted: space, tab, newline, carriage return, and =. If the extglob shell option is enabled, (will also inhibit expansion.

Several shell options settable with the shopt builtin may be used to tailor the behavior of history expansion. If the histverify shell option is enabled (see the description of the shopt builtin below), and readline is being used, history substitutions are not immediately passed to the shell parser. Instead, the expanded line is reloaded into the readline editing buffer for further modification. If readline is being used, and the histreedit shell option is enabled, a failed history substitution will be reloaded into the readline editing buffer for correction. The -p option to the history builtin command may be used to see what a history expansion will do before using it. The -s option to the history builtin may be used to add commands to the end of the history list without actually executing them, so that they are available for subsequent recall.

The shell allows control of the various characters used by the history expansion mechanism (see the description of histchars above under Shell Variables). The shell uses the history comment character to mark history timestamps when writing the history file.

Event Designators

An event designator is a reference to a command line entry in the history list. Unless the reference is absolute, events are relative to the current position in the history list.

! Start a history substitution, except when followed by a blank, newline, carriage return, = or ((when the extglob shell option is enabled using the shopt builtin).

!n Refer to command line n.

!-n Refer to the current command minus n.

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

!string

Refer to the most recent command preceding the current position in the history list starting with string.

!?string[?]

Refer to the most recent command preceding the current position in the history list containing string. The trailing ? may be

omitted if string is followed immediately by a newline. If string is missing, the string from the most recent search is used; it is an error if there is no previous search string.

^string1^string2^

Quick substitution. Repeat the previous command, replacing string1 with string2. Equivalent to ``!!:s^string1^string2^'' (see Modifiers below).

!# The entire command line typed so far.

Word Designators

Word designators are used to select desired words from the event. A : separates the event specification from the word designator. It may be omitted if the word designator begins with a ^, $, *, -, or %. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line sepa? rated by single spaces.

0 (zero)

The zeroth word. For the shell, this is the command word.

n The nth word.

^ The first argument. That is, word 1.

$ The last word. This is usually the last argument, but will ex? pand to the zeroth word if there is only one word in the line.

% The first word matched by the most recent `?string?' search, if the search string begins with a character that is part of a word.

x-y A range of words; `-y' abbreviates `0-y'.

* All of the words but the zeroth. This is a synonym for `1-$'. It is not an error to use * if there is just one word in the event; the empty string is returned in that case.

x* Abbreviates x-$.

x- Abbreviates x-$ like x*, but omits the last word. If x is miss? ing, it defaults to 0.

If a word designator is supplied without an event specification, the previous command is used as the event.

Modifiers

After the optional word designator, there may appear a sequence of one or more of the following modifiers, each preceded by a `:'. These mod? ify, or edit, the word or words selected from the history event.

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

t Remove all leading filename components, leaving the tail.

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

e Remove all but the trailing suffix.

p Print the new command but do not execute it.

q Quote the substituted words, escaping further substitutions.

x Quote the substituted words as with q, but break into words at blanks and newlines. The q and x modifiers are mutually exclu? sive; the last one supplied is used.

s/old/new/

Substitute new for the first occurrence of old in the event line. Any character may be used as the delimiter in place of /. The final delimiter is optional if it is the last character of the event line. The delimiter may be quoted in old and new with a single backslash. If & appears in new, it is replaced by old. A single backslash will quote the &. If old is null, it is set to the last old substituted, or, if no previous history substi? tutions took place, the last string in a !?string[?] search. If new is null, each matching old is deleted.

& Repeat the previous substitution.

g Cause changes to be applied over the entire event line. This is used in conjunction with `:s' (e.g., `:gs/old/new/') or `:&'. If used with `:s', any delimiter can be used in place of /, and the final delimiter is optional if it is the last character of the event line. An a may be used as a synonym for g.

G Apply the following `s' or `&' modifier once to each word in the event line.

SHELL BUILTIN COMMANDS

Unless otherwise noted, each builtin command documented in this section

as accepting options preceded by - accepts -- to signify the end of the options. The :, true, false, and test/[builtins do not accept options and do not treat -- specially. The exit, logout, return, break, con? tinue, let, and shift builtins accept and process arguments beginning with - without requiring --. Other builtins that accept arguments but are not specified as accepting options interpret arguments beginning with - as invalid options and require -- to prevent this interpreta? tion.

: [arguments]

No effect; the command does nothing beyond expanding arguments and performing any specified redirections. The return status is zero.

. filename [arguments]

source filename [arguments]

Read and execute commands from filename in the current shell en? vironment and return the exit status of the last command exe? cuted from filename. If filename does not contain a slash, filenames in PATH are used to find the directory containing filename. The file searched for in PATH need not be executable. When bash is not in posix mode, the current directory is searched if no file is found in PATH. If the sourcepath option to the shopt builtin command is turned off, the PATH is not searched. If any arguments are supplied, they become the posi? tional parameters when filename is executed. Otherwise the po? sitional parameters are unchanged. If the -T option is enabled, source inherits any trap on DEBUG; if it is not, any DEBUG trap string is saved and restored around the call to source, and source unsets the DEBUG trap while it executes. If -T is not set, and the sourced file changes the DEBUG trap, the new value is retained when source completes. The return status is the status of the last command exited within the script (0 if no commands are executed), and false if filename is not found or cannot be read.

alias [-p] [name[=value] ...]

Alias with no arguments or with the -p option prints the list of aliases in the form alias name=value on standard output. When arguments are supplied, an alias is defined for each name whose value is given. A trailing space in value causes the next word to be checked for alias substitution when the alias is expanded. For each name in the argument list for which no value is sup? plied, the name and value of the alias is printed. Alias re? turns true unless a name is given for which no alias has been defined.

bg [jobspec ...]

Resume each suspended job jobspec in the background, as if it had been started with &. If jobspec is not present, the shell's notion of the current job is used. bg jobspec returns 0 unless run when job control is disabled or, when run with job control enabled, any specified jobspec was not found or was started without job control.

bind [-m keymap] [-lpsvPSVX]

bind [-m keymap] [-q function] [-u function] [-r keyseq]

bind [-m keymap] -f filename

bind [-m keymap] -x keyseq:shell-command

bind [-m keymap] keyseq:function-name

bind [-m keymap] keyseq:readline-command

Display current readline key and function bindings, bind a key sequence to a readline function or macro, or set a readline variable. Each non-option argument is a command as it would ap? pear in .inputrc, but each binding or command must be passed as a separate argument; e.g., '"\C-x\C-r": re-read-init-file'. Op? tions, if supplied, have the following meanings:

-m keymap

Use keymap as the keymap to be affected by the subsequent bindings. Acceptable keymap names are emacs, emacs-stan? dard, emacs-meta, emacs-ctlx, vi, vi-move, vi-command,

and vi-insert. vi is equivalent to vi-command (vi-move is also a synonym); emacs is equivalent to emacs-stan? dard.

-l List the names of all readline functions.

-p Display readline function names and bindings in such a way that they can be re-read.

-P List current readline function names and bindings.

-s Display readline key sequences bound to macros and the strings they output in such a way that they can be re- read.

-S Display readline key sequences bound to macros and the strings they output.

-v Display readline variable names and values in such a way that they can be re-read.

-V List current readline variable names and values.

-f filename

Read key bindings from filename.

-q function

Query about which keys invoke the named function.

-u function

Unbind all keys bound to the named function.

-r keyseq

Remove any current binding for keyseq.

-x keyseq:shell-command

Cause shell-command to be executed whenever keyseq is en? tered. When shell-command is executed, the shell sets the READLINE_LINE variable to the contents of the read? line line buffer and the READLINE_POINT and READLINE_MARK variables to the current location of the insertion point and the saved insertion point (the mark), respectively. If the executed command changes the value of any of READ? LINE_LINE, READLINE_POINT, or READLINE_MARK, those new values will be reflected in the editing state.

-X List all key sequences bound to shell commands and the associated commands in a format that can be reused as in? put.

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

break [n]

Exit from within a for, while, until, or select 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 n is not greater than or equal to 1.

builtin shell-builtin [arguments]

Execute the specified shell builtin, passing it arguments, and return its exit status. This is useful when defining a function whose name is the same as a shell builtin, retaining the func? tionality of the builtin within the function. The cd builtin is commonly redefined this way. The return status is false if shell-builtin is not a shell builtin command.

caller [expr]

Returns the context of any active subroutine call (a shell func? tion or a script executed with the . or source builtins). With? out expr, caller displays the line number and source filename of the current subroutine call. If a non-negative integer is sup? plied as expr, caller displays the line number, subroutine name, and source file corresponding to that position in the current execution call stack. This extra information may be used, for example, to print a stack trace. The current frame is frame 0. The return value is 0 unless the shell is not executing a sub? routine call or expr does not correspond to a valid position in the call stack.

cd [-L|[-P [-e]] [-@]] [dir]

Change the current directory to dir. if dir is not supplied, the value of the HOME shell variable is the default. Any addi?

tional arguments following dir are ignored. The variable CDPATH defines the search path for the directory containing dir: each directory name in CDPATH is searched for dir. Alternative di? rectory names in CDPATH 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. The -P option causes cd to use the physical directory structure by resolving symbolic links while traversing dir and before processing instances of .. in dir (see also the -P option to the set builtin command); the -L option forces symbolic links to be followed by resolving the link after processing instances of .. in dir. If .. appears in dir, it is processed by removing the immediately previous pathname component from dir, back to a slash or the beginning of dir. If the -e option is supplied with -P, and the current working directory cannot be success? fully determined after a successful directory change, cd will return an unsuccessful status. On systems that support it, the -@ option presents the extended attributes associated with a file as a directory. An argument of - is converted to $OLDPWD before the directory change is attempted. If a non-empty direc? tory name from CDPATH is used, or if - is the first argument, and the directory change is successful, the absolute pathname of the new working directory is written to the standard output. The return value is true if the directory was successfully changed; false otherwise.

command [-pVv] command [arg ...]

Run command with args suppressing the normal shell function lookup. Only builtin commands or commands found in the PATH are executed. If the -p option is given, the search for command is performed using a default value for PATH that is guaranteed to find all of the standard utilities. If either the -V or -v op? tion is supplied, a description of command is printed. The -v option causes a single word indicating the command or filename

used to invoke command to be displayed; the -V option produces a more verbose description. If the -V or -v option is supplied, the exit status is 0 if command was found, and 1 if not. If neither option is supplied and an error occurred or command can? not be found, the exit status is 127. Otherwise, the exit sta? tus of the command builtin is the exit status of command.

compgen [option] [word]

Generate possible completion matches for word according to the options, which may be any option accepted by the complete builtin with the exception of -p and -r, and write the matches to the standard output. When using the -F or -C options, the various shell variables set by the programmable completion fa? cilities, while available, will not have useful values.

The matches will be generated in the same way as if the program? mable completion code had generated them directly from a comple? tion specification with the same flags. If word is specified, only those completions matching word will be displayed.

The return value is true unless an invalid option is supplied, or no matches were generated.

complete [-abcdefgjksuv] [-o comp-option] [-DEI] [-A action] [-G glob? pat] [-W wordlist]

[-F function] [-C command] [-X filterpat] [-P prefix] [-S suf? fix] name [name ...]

complete -pr [-DEI] [name ...]

Specify how arguments to each name should be completed. If the -p option is supplied, or if no options are supplied, existing completion specifications are printed in a way that allows them to be reused as input. The -r option removes a completion spec? ification for each name, or, if no names are supplied, all com? pletion specifications. The -D option indicates that other sup? plied options and actions should apply to the ``default'' com? mand completion; that is, completion attempted on a command for which no completion has previously been defined. The -E option

indicates that other supplied options and actions should apply to ``empty'' command completion; that is, completion attempted on a blank line. The -I option indicates that other supplied options and actions should apply to completion on the initial non-assignment word on the line, or after a command delimiter such as ; or |, which is usually command name completion. If multiple options are supplied, the -D option takes precedence over -E, and both take precedence over -I. If any of -D, -E, or -I are supplied, any other name arguments are ignored; these completions only apply to the case specified by the option.

The process of applying these completion specifications when word completion is attempted is described above under Program? mable Completion.

Other options, if specified, have the following meanings. The arguments to the -G, -W, and -X options (and, if necessary, the -P and -S options) should be quoted to protect them from expan? sion before the complete builtin is invoked.

-o comp-option

The comp-option controls several aspects of the comp? spec's behavior beyond the simple generation of comple? tions. comp-option may be one of:

bashdefault

Perform the rest of the default bash completions if the compspec generates no matches.

default Use readline's default filename completion if the compspec generates no matches.

dirnames

Perform directory name completion if the comp? spec generates no matches.

filenames

Tell readline that the compspec generates file? names, so it can perform any filename-specific processing (like adding a slash to directory

names, quoting special characters, or suppress?
ing trailing spaces). Intended to be used with
shell functions.

noquote Tell readline not to quote the completed words
if they are filenames (quoting filenames is the
default).

nosort Tell readline not to sort the list of possible
completions alphabetically.

nospace Tell readline not to append a space (the de?
fault) to words completed at the end of the
line.

plusdirs
After any matches defined by the compspec are
generated, directory name completion is at?
tempted and any matches are added to the results
of the other actions.

-A action
The action may be one of the following to generate a
list of possible completions:

alias Alias names. May also be specified as -a.

arrayvar
Array variable names.

binding Readline key binding names.

builtin Names of shell builtin commands. May also be
specified as -b.

command Command names. May also be specified as -c.

directory
Directory names. May also be specified as -d.

disabled
Names of disabled shell builtins.

enabled Names of enabled shell builtins.

export Names of exported shell variables. May also be
specified as -e.

file File names. May also be specified as -f.

function

Names of shell functions.

group Group names. May also be specified as -g.

helptopic

Help topics as accepted by the help builtin.

hostname

Hostnames, as taken from the file specified by the HOSTFILE shell variable.

job Job names, if job control is active. May also be specified as -j.

keyword Shell reserved words. May also be specified as -k.

running Names of running jobs, if job control is active.

service Service names. May also be specified as -s.

setopt Valid arguments for the -o option to the set builtin.

shopt Shell option names as accepted by the shopt builtin.

signal Signal names.

stopped Names of stopped jobs, if job control is active.

user User names. May also be specified as -u.

variable

Names of all shell variables. May also be spec? ified as -v.

-C command

command is executed in a subshell environment, and its output is used as the possible completions.

-F function

The shell function function is executed in the current shell environment. When the function is executed, the first argument ($1) is the name of the command whose ar? guments are being completed, the second argument ($2) is

the word being completed, and the third argument ($3) is the word preceding the word being completed on the cur? rent command line. When it finishes, the possible com? pletions are retrieved from the value of the COMPREPLY array variable.

-G globpat

The pathname expansion pattern globpat is expanded to generate the possible completions.

-P prefix

prefix is added at the beginning of each possible com? pletion after all other options have been applied.

-S suffix

suffix is appended to each possible completion after all other options have been applied.

-W wordlist

The wordlist is split using the characters in the IFS special variable as delimiters, and each resultant word is expanded. Shell quoting is honored within wordlist, in order to provide a mechanism for the words to contain shell metacharacters or characters in the value of IFS. The possible completions are the members of the resul? tant list which match the word being completed.

-X filterpat

filterpat is a pattern as used for pathname expansion. It is applied to the list of possible completions gener? ated by the preceding options and arguments, and each completion matching filterpat is removed from the list. A leading ! in filterpat negates the pattern; in this case, any completion not matching filterpat is removed.

The return value is true unless an invalid option is supplied, an option other than -p or -r is supplied without a name argu? ment, an attempt is made to remove a completion specification for a name for which no specification exists, or an error occurs

adding a completion specification.

compopt [-o option] [-DEI] [+o option] [name]

Modify completion options for each name according to the op? tions, or for the currently-executing completion if no names are supplied. If no options are given, display the completion op? tions for each name or the current completion. The possible values of option are those valid for the complete builtin de? scribed above. The -D option indicates that other supplied op? tions should apply to the ``default'' command completion; that is, completion attempted on a command for which no completion has previously been defined. The -E option indicates that other supplied options should apply to ``empty'' command completion; that is, completion attempted on a blank line. The -I option indicates that other supplied options should apply to completion on the initial non-assignment word on the line, or after a com? mand delimiter such as ; or |, which is usually command name completion.

The return value is true unless an invalid option is supplied, an attempt is made to modify the options for a name for which no completion specification exists, or an output error occurs.

continue [n]

Resume the next iteration of the enclosing for, while, until, or select loop. If n is specified, resume at the nth enclosing loop. n must be ? 1. If n is greater than the number of en? closing loops, the last enclosing loop (the ``top-level'' loop) is resumed. The return value is 0 unless n is not greater than or equal to 1.

declare [-aAfFgiIlnrtux] [-p] [name[=value] ...]

typeset [-aAfFgiIlnrtux] [-p] [name[=value] ...]

Declare variables and/or give them attributes. If no names are given then display the values of variables. The -p option will display the attributes and values of each name. When -p is used with name arguments, additional options, other than -f and -F,

are ignored. When -p is supplied without name arguments, it will display the attributes and values of all variables having the attributes specified by the additional options. If no other options are supplied with -p, declare will display the at? tributes and values of all shell variables. The -f option will restrict the display to shell functions. The -F option inhibits the display of function definitions; only the function name and attributes are printed. If the extdebug shell option is enabled using shopt, the source file name and line number where each name is defined are displayed as well. The -F option implies -f. The -g option forces variables to be created or modified at the global scope, even when declare is executed in a shell func? tion. It is ignored in all other cases. The -I option causes local variables to inherit the attributes (except the nameref attribute) and value of any existing variable with the same name at a surrounding scope. If there is no existing variable, the local variable is initially unset. The following options can be used to restrict output to variables with the specified attri? bute or to give variables attributes:

-a Each name is an indexed array variable (see Arrays above).

-A Each name is an associative array variable (see Arrays above).

-f Use function names only.

-i The variable is treated as an integer; arithmetic evalua? tion (see ARITHMETIC EVALUATION above) is performed when the variable is assigned a value.

-l When the variable is assigned a value, all upper-case characters are converted to lower-case. The upper-case attribute is disabled.

-n Give each name the nameref attribute, making it a name reference to another variable. That other variable is defined by the value of name. All references, assign?

ments, and attribute modifications to name, except those using or changing the -n attribute itself, are performed on the variable referenced by name's value. The nameref attribute cannot be applied to array variables.

-r Make names readonly. These names cannot then be assigned values by subsequent assignment statements or unset.

-t Give each name the trace attribute. Traced functions inherit the DEBUG and RETURN traps from the calling shell. The trace attribute has no special meaning for variables.

-u When the variable is assigned a value, all lower-case characters are converted to upper-case. The lower-case attribute is disabled.

-x Mark names for export to subsequent commands via the environment.

Using `+' instead of `-' turns off the attribute instead, with the exceptions that +a and +A may not be used to destroy array variables and +r will not remove the readonly attribute. When used in a function, declare and typeset make each name local, as with the local command, unless the -g option is supplied. If a variable name is followed by =value, the value of the variable is set to value. When using -a or -A and the compound assignment syntax to create array variables, additional attributes do not take effect until subsequent assignments. The return value is 0 unless an invalid option is encountered, an attempt is made to define a function using ``-f foo=bar'', an attempt is made to assign a value to a readonly variable, an attempt is made to assign a value to an array variable without using the compound assignment syntax (see Arrays above), one of the names is not a valid shell variable name, an attempt is made to turn off readonly status for a readonly variable, an attempt is made to turn off array status for an array variable, or an attempt is made to display a non-existent function with -f.

Without options, displays the list of currently remembered di? rectories. The default display is on a single line with direc? tory names separated by spaces. Directories are added to the list with the pushd command; the popd command removes entries from the list. The current directory is always the first direc? tory in the stack.

-c Clears the directory stack by deleting all of the en? tries.

-l Produces a listing using full pathnames; the default listing format uses a tilde to denote the home directory.

-p Print the directory stack with one entry per line.

-v Print the directory stack with one entry per line, pre? fixing each entry with its index in the stack.

+n Displays the nth entry counting from the left of the list shown by dirs when invoked without options, starting with zero.

-n Displays the nth entry counting from the right of the list shown by dirs when invoked without options, starting with zero.

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

disown [-ar] [-h] [jobspec ... | pid ...]

Without options, remove each jobspec from the table of active jobs. If jobspec is not present, and neither the -a nor the -r option is supplied, the current job is used. If the -h option is given, each jobspec is not removed from the table, but is marked so that SIGHUP is not sent to the job if the shell re? ceives a SIGHUP. If no jobspec is supplied, the -a option means to remove or mark all jobs; the -r option without a jobspec ar? gument restricts operation to running jobs. The return value is 0 unless a jobspec does not specify a valid job.

echo [-neE] [arg ...]

Output the args, separated by spaces, followed by a newline.

The return status is 0 unless a write error occurs. If -n is specified, the trailing newline is suppressed. If the -e option is given, interpretation of the following backslash-escaped characters is enabled. The -E option disables the interpreta? tion of these escape characters, even on systems where they are interpreted by default. The xpg_echo shell option may be used to dynamically determine whether or not echo expands these es? cape characters by default. echo does not interpret -- to mean the end of options. echo interprets the following escape se? quences:

\a alert (bell)

\b backspace

\c suppress further output

\e

\E an escape character

\f form feed

\n new line

\r carriage return

\t horizontal tab

\v vertical tab

\\ backslash

\0nnn the eight-bit character whose value is the octal value nnn (zero to three octal digits)

\xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

\uHHHH the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value HHHH (one to four hex digits)

\UHHHHHHHH

the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value HHHHHHHH (one to eight hex digits)

enable [-a] [-dnps] [-f filename] [name ...]

Enable and disable builtin shell commands. Disabling a builtin allows a disk command which has the same name as a shell builtin

to be executed without specifying a full pathname, even though the shell normally searches for builtins before disk commands. If -n is used, each name is disabled; otherwise, names are en? abled. For example, to use the test binary found via the PATH instead of the shell builtin version, run ``enable -n test''. The -f option means to load the new builtin command name from shared object filename, on systems that support dynamic loading. The -d option will delete a builtin previously loaded with -f. If no name arguments are given, or if the -p option is supplied, a list of shell builtins is printed. With no other option argu? ments, the list consists of all enabled shell builtins. If -n is supplied, only disabled builtins are printed. If -a is sup? plied, the list printed includes all builtins, with an indica? tion of whether or not each is enabled. If -s is supplied, the output is restricted to the POSIX special builtins. The return value is 0 unless a name is not a shell builtin or there is an error loading a new builtin from a shared object.

eval [arg ...]

The args are read and concatenated together into a single com? mand. This command is then read and executed by the shell, and its exit status is returned as the value of eval. If there are no args, or only null arguments, eval returns 0.

exec [-cl] [-a name] [command [arguments]]

If command is specified, it replaces the shell. No new process is created. The arguments become the arguments to command. If the -l option is supplied, the shell places a dash at the begin? ning of the zeroth argument passed to command. This is what lo? gin(1) does. The -c option causes command to be executed with an empty environment. If -a is supplied, the shell passes name as the zeroth argument to the executed command. If command can? not be executed for some reason, a non-interactive shell exits, unless the execfail shell option is enabled. In that case, it returns failure. An interactive shell returns failure if the

file cannot be executed. A subshell exits unconditionally if exec fails. If command is not specified, any redirections take effect in the current shell, and the return status is 0. If there is a redirection error, the return status is 1.

exit [n]

Cause the shell to exit with a status of n. If n is omitted, the exit status is that of the last command executed. A trap on EXIT is executed before the shell terminates.

export [-fn] [name[=word]] ...

export -p

The supplied names are marked for automatic export to the envi? ronment of subsequently executed commands. If the -f option is given, the names refer to functions. If no names are given, or if the -p option is supplied, a list of names of all exported variables is printed. The -n option causes the export property to be removed from each name. If a variable name is followed by =word, the value of the variable is set to word. export returns an exit status of 0 unless an invalid option is encountered, one of the names is not a valid shell variable name, or -f is sup? plied with a name that is not a function.

fc [-e ename] [-lnr] [first] [last]

fc -s [pat=rep] [cmd]

The first form selects a range of commands from first to last from the history list and displays or edits and re-executes them. First and last may be specified as a string (to locate the last command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). When listing, a first or last of 0 is equivalent to -1 and -0 is equivalent to the current command (usually the fc command); otherwise 0 is equivalent to -1 and -0 is invalid. If last is not specified, it is set to the current command for listing (so that ``fc -l -10'' prints the last 10 commands) and to first otherwise. If

first is not specified, it is set to the previous command for editing and -16 for listing.

The -n option suppresses the command numbers when listing. The -r option reverses the order of the commands. If the -l option is given, the commands are listed on standard output. Other? wise, 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 com? plete, the edited commands are echoed and executed.

In the second form, command is re-executed after each instance of pat is replaced by rep. Command is interpreted the same as first above. A useful alias to use with this is ``r="fc -s"'', so that typing ``r cc'' runs the last command beginning with ``cc'' and typing ``r'' re-executes the last command.

If the first form is used, the return value is 0 unless an in? valid option is encountered or first or last specify history lines out of range. If the -e option is supplied, the return value is the value of the last command executed or failure if an error occurs with the temporary file of commands. If the second form is used, the return status is that of the command re-exe? cuted, unless cmd does not specify a valid history line, in which case fc returns failure.

fg [jobspec]

Resume jobspec in the foreground, and make it the current job. If jobspec is not present, the shell's notion of the current job is used. The return value is that of the command placed into the foreground, or failure if run when job control is disabled or, when run with job control enabled, if jobspec does not spec? ify a valid job or jobspec specifies a job that was started without job control.

getopts optstring name [arg ...]

getopts is used by shell procedures to parse positional parame?

ters. optstring contains the option characters to be recog?
nized; if a character is followed by a colon, the option is ex?
pected to have an argument, which should be separated from it by
white space. The colon and question mark characters may not be
used as option characters. Each time it is invoked, getopts
places the next option in the shell variable name, initializing
name if it does not exist, and the index of the next argument to
be processed into the variable OPTIND. OPTIND is initialized to
1 each time the shell or a shell script is invoked. When an op?
tion 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.

When the end of options is encountered, getopts exits with a re?
turn value greater than zero. OPTIND is set to the index of the
first non-option argument, and name is set to ?.

getopts normally parses the positional parameters, but if more
arguments are supplied as arg values, getopts parses those in?
stead.

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 invalid
options or missing option arguments are encountered. If the
variable OPTERR is set to 0, no error messages will be dis?
played, even if the first character of optstring is not a colon.

If an invalid 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 OP?
TARG 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 returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

hash [-lr] [-p filename] [-dt] [name]

Each time hash is invoked, the full pathname of the command name is determined by searching the directories in $PATH and remem? bered. Any previously-remembered pathname is discarded. If the -p option is supplied, no path search is performed, and filename is used as the full filename of the command. The -r option causes the shell to forget all remembered locations. The -d op? tion causes the shell to forget the remembered location of each name. If the -t option is supplied, the full pathname to which each name corresponds is printed. If multiple name arguments are supplied with -t, the name is printed before the hashed full pathname. The -l option causes output to be displayed in a for? mat that may be reused as input. If no arguments are given, or if only -l is supplied, information about remembered commands is printed. The return status is true unless a name is not found or an invalid option is supplied.

help [-dms] [pattern]

Display helpful information about builtin commands. If pattern is specified, help gives detailed help on all commands matching pattern; otherwise help for all the builtins and shell control structures is printed.

-d Display a short description of each pattern

-m Display the description of each pattern in a manpage-like format

-s Display only a short usage synopsis for each pattern

The return status is 0 unless no command matches pattern.

history [n]

history -c

history -d offset

history -d start-end

history -anrw [filename]

history -p arg [arg ...]

history -s arg [arg ...]

With no options, display the command history list with line num? bers. Lines listed with a * have been modified. An argument of n lists only the last n lines. If the shell variable HISTTIME? FORMAT is set and not null, it is used as a format string for strftime(3) to display the time stamp associated with each dis? played history entry. No intervening blank is printed between the formatted time stamp and the history line. If filename is supplied, it is used as the name of the history file; if not, the value of HISTFILE is used. Options, if supplied, have the following meanings:

-c Clear the history list by deleting all the entries.

-d offset

Delete the history entry at position offset. If offset is negative, it is interpreted as relative to one greater than the last history position, so negative indices count back from the end of the history, and an index of -1 refers to the current history -d command.

-d start-end

Delete the history entries between positions start and end, inclusive. Positive and negative values for start and end are interpreted as described above.

-a Append the ``new'' history lines to the history file. These are history lines entered since the beginning of the current bash session, but not already appended to the history file.

-n Read the history lines not already read from the history file into the current history list. These are lines ap? pended to the history file since the beginning of the

current bash session.

-r Read the contents of the history file and append them to the current history list.

-w Write the current history list to the history file, over? writing the history file's contents.

-p Perform history substitution on the following args and display the result on the standard output. Does not store the results in the history list. Each arg must be quoted to disable normal history expansion.

-s Store the args in the history list as a single entry. The last command in the history list is removed before the args are added.

If the HISTTIMEFORMAT variable is set, the time stamp informa? tion associated with each history entry is written to the his? tory file, marked with the history comment character. When the history file is read, lines beginning with the history comment character followed immediately by a digit are interpreted as timestamps for the following history entry. The return value is 0 unless an invalid option is encountered, an error occurs while reading or writing the history file, an invalid offset is sup? plied as an argument to -d, or the history expansion supplied as an argument to -p fails.

jobs [-lnprs] [jobspec ...]

jobs -x command [args ...]

The first form lists the active jobs. The options have the fol? lowing meanings:

-l List process IDs in addition to the normal information.

-n Display information only about jobs that have changed status since the user was last notified of their status.

-p List only the process ID of the job's process group leader.

-r Display only running jobs.

-s Display only stopped jobs.

If jobspec is given, output is restricted to information about that job. The return status is 0 unless an invalid option is encountered or an invalid jobspec is supplied.

If the -x option is supplied, jobs replaces any jobspec found in command or args with the corresponding process group ID, and ex? ecutes command passing it args, returning its exit status.

kill [-s sigspec | -n signum | -sigspec] [pid | jobspec] ...

kill -l|-L [sigspec | exit_status]

Send the signal named by sigspec or signum to the processes named by pid or jobspec. sigspec is either a case-insensitive signal name such as SIGKILL (with or without the SIG prefix) or a signal number; signum is a signal number. If sigspec is not 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 signals corresponding to the arguments are listed, and the return status is 0. The exit_status argument to -l is a number specifying either a signal number or the exit status of a process terminated by a signal. The -L option is equivalent to -l. kill returns true if at least one signal was successfully sent, or false if an error occurs or an invalid op? tion is encountered.

let arg [arg ...]

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

local [option] [name[=value] ... | -]

For each argument, a local variable named name is created, and assigned value. The option can be any of the options accepted by declare. When local is used within a function, it causes the variable name to have a visible scope restricted to that func? tion and its children. If name is -, the set of shell options is made local to the function in which local is invoked: shell options changed using the set builtin inside the function are

restored to their original values when the function returns. The restore is effected as if a series of set commands were exe? cuted to restore the values that were in place before the func? tion. With no operands, local writes a list of local variables to the standard output. It is an error to use local when not within a function. The return status is 0 unless local is used outside a function, an invalid name is supplied, or name is a readonly variable.

logout Exit a login shell.

mapfile [-d delim] [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]

readarray [-d delim] [-n count] [-O origin] [-s count] [-t] [-u fd] [-C callback] [-c quantum] [array]

Read lines from the standard input into the indexed array vari? able array, or from file descriptor fd if the -u option is sup? plied. The variable MAPFILE is the default array. Options, if supplied, have the following meanings:

-d The first character of delim is used to terminate each input line, rather than newline. If delim is the empty string, mapfile will terminate a line when it reads a NUL character.

-n Copy at most count lines. If count is 0, all lines are copied.

-O Begin assigning to array at index origin. The default index is 0.

-s Discard the first count lines read.

-t Remove a trailing delim (default newline) from each line read.

-u Read lines from file descriptor fd instead of the stan? dard input.

-C Evaluate callback each time quantum lines are read. The -c option specifies quantum.

-c Specify the number of lines read between each call to

callback.

If -C is specified without -c, the default quantum is 5000.

When callback is evaluated, it is supplied the index of the next array element to be assigned and the line to be assigned to that element as additional arguments. callback is evaluated after the line is read but before the array element is assigned.

If not supplied with an explicit origin, mapfile will clear ar? ray before assigning to it.

mapfile returns successfully unless an invalid option or option argument is supplied, array is invalid or unassignable, or if array is not an indexed array.

popd [-n] [+n] [-n]

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. Arguments, if supplied, have the follow? ing meanings:

-n Suppresses the normal change of directory when removing directories from the stack, so that only the stack is ma? nipulated.

+n Removes the nth 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 nth entry counting from the right of the list shown by dirs, starting with zero. For example: ``popd -0'' removes the last directory, ``popd -1'' the next to last.

If the popd command is successful, a dirs is performed as well, and the return status is 0. popd returns false if an invalid option is encountered, the directory stack is empty, a non-exis? tent directory stack entry is specified, or the directory change fails.

printf [-v var] format [arguments]

Write the formatted arguments to the standard output under the

control of the format. The -v option causes the output to be assigned to the variable var rather than being printed to the standard output.

The format is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences, which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive argument. In addition to the standard printf(1) format specifications, printf interprets the following extensions:

%b causes printf to expand backslash escape sequences in the corresponding argument in the same way as echo -e.

%q causes printf to output the corresponding argument in a format that can be reused as shell input.

%(datefmt)T

causes printf to output the date-time string resulting from using datefmt as a format string for strftime(3). The corresponding argument is an integer representing the number of seconds since the epoch. Two special argument values may be used: -1 represents the current time, and -2 represents the time the shell was invoked. If no ar? gument is specified, conversion behaves as if -1 had been given. This is an exception to the usual printf behav? ior.

The %b, %q, and %T directives all use the field width and preci? sion arguments from the format specification and write that many bytes from (or use that wide a field for) the expanded argument, which usually contains more characters than the original.

Arguments to non-string format specifiers are treated as C con? stants, except that a leading plus or minus sign is allowed, and if the leading character is a single or double quote, the value is the ASCII value of the following character.

The format is reused as necessary to consume all of the argu?

ments. If the format requires more arguments than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied. The return value is zero on success, non-zero on failure.

pushd [-n] [+n] [-n]

pushd [-n] [dir]

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, pushd exchanges the top two di? rectories and returns 0, unless the directory stack is empty. Arguments, if supplied, have the following meanings:

-n Suppresses the normal change of directory when rotating or adding directories to the stack, so that only the stack is manipulated.

+n Rotates the stack so that the nth directory (counting from the left of the list shown by dirs, starting with zero) is at the top.

-n Rotates the stack so that the nth directory (counting from the right of the list shown by dirs, starting with zero) is at the top.

dir Adds dir to the directory stack at the top, making it the new current working directory as if it had been supplied as the argument to the cd builtin.

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

pwd [-LP]

Print the absolute pathname of the current working directory. The pathname printed contains no symbolic links if the -P option is supplied or the -o physical option to the set builtin command

is enabled. If the -L option is used, the pathname printed may contain symbolic links. The return status is 0 unless an error occurs while reading the name of the current directory or an in? valid option is supplied.

read [-ers] [-a aname] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]

One line is read from the standard input, or from the file de? scriptor fd supplied as an argument to the -u option, split into words as described above under Word Splitting, and the first word is assigned to the first name, the second word to the sec? ond name, and so on. If there are more words than names, the remaining words and their intervening delimiters are assigned to the last name. If there are fewer words read from the input stream than names, the remaining names are assigned empty val? ues. The characters in IFS are used to split the line into words using the same rules the shell uses for expansion (de? scribed above under Word Splitting). The backslash character (\) may be used to remove any special meaning for the next char? acter read and for line continuation. Options, if supplied, have the following meanings:

-a aname

The words are assigned to sequential indices of the array variable aname, starting at 0. aname is unset before any new values are assigned. Other name arguments are ig? nored.

-d delim

The first character of delim is used to terminate the in? put line, rather than newline. If delim is the empty string, read will terminate a line when it reads a NUL character.

-e If the standard input is coming from a terminal, readline (see READLINE above) is used to obtain the line. Read? line uses the current (or default, if line editing was

not previously active) editing settings, but uses Read? line's default filename completion.

-i text

If readline is being used to read the line, text is placed into the editing buffer before editing begins.

-n nchars

read returns after reading nchars characters rather than waiting for a complete line of input, but honors a delim? iter if fewer than nchars characters are read before the delimiter.

-N nchars

read returns after reading exactly nchars characters rather than waiting for a complete line of input, unless EOF is encountered or read times out. Delimiter charac? ters encountered in the input are not treated specially and do not cause read to return until nchars characters are read. The result is not split on the characters in IFS; the intent is that the variable is assigned exactly the characters read (with the exception of backslash; see the -r option below).

-p prompt

Display prompt on standard error, without a trailing new? line, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.

-r Backslash does not act as an escape character. The back? slash is considered to be part of the line. In particu? lar, a backslash-newline pair may not then be used as a line continuation.

-s Silent mode. If input is coming from a terminal, charac? ters are not echoed.

-t timeout

Cause read to time out and return failure if a complete line of input (or a specified number of characters) is

not read within timeout seconds. timeout may be a deci?
mal number with a fractional portion following the deci?
mal point. This option is only effective if read is
reading input from a terminal, pipe, or other special
file; it has no effect when reading from regular files.
If read times out, read saves any partial input read into
the specified variable name. If timeout is 0, read re?
turns immediately, without trying to read any data. The
exit status is 0 if input is available on the specified
file descriptor, non-zero otherwise. The exit status is
greater than 128 if the timeout is exceeded.

-u fd Read input from file descriptor fd.

If no names are supplied, the line read, without the ending de?
limiter but otherwise unmodified, is assigned to the variable
REPLY. The exit status is zero, unless end-of-file is encoun?
tered, read times out (in which case the status is greater than
128), a variable assignment error (such as assigning to a read?
only variable) occurs, or an invalid file descriptor is supplied
as the argument to -u.

readonly [-aAf] [-p] [name[=word] ...]

The given names are marked readonly; the values of these names
may not be changed by subsequent assignment. If the -f option
is supplied, the functions corresponding to the names are so
marked. The -a option restricts the variables to indexed ar?
rays; the -A option restricts the variables to associative ar?
rays. If both options are supplied, -A takes precedence. If no
name arguments are given, or if the -p option is supplied, a
list of all readonly names is printed. The other options may be
used to restrict the output to a subset of the set of readonly
names. The -p option causes output to be displayed in a format
that may be reused as input. If a variable name is followed by
=word, the value of the variable is set to word. The return
status is 0 unless an invalid option is encountered, one of the

names is not a valid shell variable name, or -f is supplied with a name that is not a function.

return [n]

Causes a function to stop executing and return the value speci? fied by n to its caller. If n is omitted, the return status is that of the last command executed in the function body. If re? turn is executed by a trap handler, the last command used to de? termine the status is the last command executed before the trap handler. If return is executed during a DEBUG trap, the last command used to determine the status is the last command exe? cuted by the trap handler before return was invoked. If return is used outside a function, but during execution of a script by the . (source) command, it causes the shell to stop executing that script and return either n or the exit status of the last command executed within the script as the exit status of the script. If n is supplied, the return value is its least signif? icant 8 bits. The return status is non-zero if return is sup? plied a non-numeric argument, or is used outside a function and not during execution of a script by . or source. Any command associated with the RETURN trap is executed before execution re? sumes after the function or script.

set [--abefhkmnptuvxBCEHPT] [-o option-name] [arg ...]

set [+abefhkmnptuvxBCEHPT] [+o option-name] [arg ...]

Without options, the name and value of each shell variable are displayed in a format that can be reused as input for setting or resetting the currently-set variables. Read-only variables can? not be reset. In posix mode, only shell variables are listed. The output is sorted according to the current locale. When op? tions are specified, they set or unset shell attributes. Any arguments remaining after option processing are treated as val? ues for the positional parameters and are assigned, in order, to $1, $2, ... $n. Options, if specified, have the following meanings:

-a Each variable or function that is created or modified is given the export attribute and marked for export to the environment of subsequent commands.

-b Report the status of terminated background jobs immedi? ately, rather than before the next primary prompt. This is effective only when job control is enabled.

-e Exit immediately if a pipeline (which may consist of a single simple command), a list, or a compound command (see SHELL GRAMMAR above), exits with a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a while or until keyword, part of the test following the if or elif reserved words, part of any command executed in a && or || list except the command following the final && or ||, any command in a pipeline but the last, or if the command's return value is being inverted with !. If a compound command other than a subshell returns a non- zero status because a command failed while -e was being ignored, the shell does not exit. A trap on ERR, if set, is executed before the shell exits. This option applies to the shell environment and each subshell envi? ronment separately (see COMMAND EXECUTION ENVIRONMENT above), and may cause subshells to exit before executing all the commands in the subshell.

If a compound command or shell function executes in a context where -e is being ignored, none of the commands executed within the compound command or function body will be affected by the -e setting, even if -e is set and a command returns a failure status. If a compound command or shell function sets -e while executing in a context where -e is ignored, that setting will not have any effect until the compound command or the command containing the function call completes.

-f Disable pathname expansion.

-h Remember the location of commands as they are looked up for execution. This is enabled by default.

-k All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.

-m Monitor mode. Job control is enabled. This option is on by default for interactive shells on systems that support it (see JOB CONTROL above). All processes run in a separate process group. When a background job com? pletes, the shell prints a line containing its exit sta? tus.

-n Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ig? nored by interactive shells.

-o option-name

The option-name can be one of the following:

allexport

Same as -a.

braceexpand

Same as -B.

emacs Use an emacs-style command line editing inter? face. This is enabled by default when the shell is interactive, unless the shell is started with the --noediting option. This also affects the editing interface used for read -e.

errexit Same as -e.

errtrace

Same as -E.

functrace

Same as -T.

hashall Same as -h.

histexpand

Same as -H.

history Enable command history, as described above under HISTORY. This option is on by default in inter? active shells.

ignoreeof

The effect is as if the shell command ``IG? NOREEOF=10" had been executed (see Shell Vari? ables above).

keyword Same as -k.

monitor Same as -m.

noclobber

Same as -C.

noexec Same as -n.

noglob Same as -f.

nolog Currently ignored.

notify Same as -b.

nounset Same as -u.

onecmd Same as -t.

physical

Same as -P.

pipefail

If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

posix Change the behavior of bash where the default operation differs from the POSIX standard to match the standard (posix mode). See SEE ALSO below for a reference to a document that details how posix mode affects bash's behavior.

privileged

Same as -p.

verbose Same as -v.

vi Use a vi-style command line editing interface.
This also affects the editing interface used for
read -e.

xtrace Same as -x.

If -o is supplied with no option-name, the values of the current options are printed. If +o is supplied with no option-name, a series of set commands to recreate the current option settings is displayed on the standard output.

-p Turn on privileged mode. In this mode, the $ENV and $BASH_ENV files are not processed, shell functions are not inherited from the environment, and the SHELLOPTS, BASHOPTS, CDPATH, and GLOBIGNORE variables, if they ap? pear in the environment, are ignored. If the shell is started with the effective user (group) id not equal to the real user (group) id, and the -p option is not sup? plied, these actions are taken and the effective user id is set to the real user id. If the -p option is sup? plied at startup, the effective user id is not reset. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

-t Exit after reading and executing one command.

-u Treat unset variables and parameters other than the spe? cial parameters "@" and "*" as an error when performing parameter expansion. If expansion is attempted on an unset variable or parameter, the shell prints an error message, and, if not interactive, exits with a non-zero status.

-v Print shell input lines as they are read.

-x After expanding each simple command, for command, case command, select command, or arithmetic for command, dis? play the expanded value of PS4, followed by the command

and its expanded arguments or associated word list.

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

-C If set, bash does not overwrite an existing file with the >, >&, and <> redirection operators. This may be overridden when creating output files by using the redi? rection operator >| instead of >.

-E If set, any trap on ERR is inherited by shell functions, command substitutions, and commands executed in a sub? shell environment. The ERR trap is normally not inher? ited in such cases.

-H Enable ! style history substitution. This option is on by default when the shell is interactive.

-P If set, the shell does not resolve symbolic links when executing commands such as cd that change the current working directory. It uses the physical directory structure instead. By default, bash follows the logical chain of directories when performing commands which change the current directory.

-T If set, any traps on DEBUG and RETURN are inherited by shell functions, command substitutions, and commands ex? ecuted in a subshell environment. The DEBUG and RETURN traps are normally not inherited in such cases.

-- If no arguments follow this option, then the positional parameters are unset. Otherwise, the positional parame? ters 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 posi? tional parameters remain unchanged.

The options are off by default unless otherwise noted. Using + rather than - causes these options to be turned off. The op?

tions can also be specified as arguments to an invocation of the shell. The current set of options may be found in $-. The re? turn status is always true unless an invalid option is encoun? tered.

shift [n]

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

shopt [-pqsu] [-o] [optname ...]

Toggle the values of settings controlling optional shell behav? ior. The settings can be either those listed below, or, if the -o option is used, those available with the -o option to the set builtin command. With no options, or with the -p option, a list of all settable options is displayed, with an indication of whether or not each is set; if optnames are supplied, the output is restricted to those options. The -p option causes output to be displayed in a form that may be reused as input. Other op? tions have the following meanings:

-s Enable (set) each optname.

-u Disable (unset) each optname.

-q Suppresses normal output (quiet mode); the return status indicates whether the optname is set or unset. If multi? ple optname arguments are given with -q, the return sta? tus is zero if all optnames are enabled; non-zero other? wise.

-o Restricts the values of optname to be those defined for the -o option to the set builtin.

If either -s or -u is used with no optname arguments, shopt shows only those options which are set or unset, respectively.

Unless otherwise noted, the shopt options are disabled (unset) by default.

The return status when listing options is zero if all optnames are enabled, non-zero otherwise. When setting or unsetting op? tions, the return status is zero unless an optname is not a valid shell option.

The list of shopt options is:

assoc_expand_once

If set, the shell suppresses multiple evaluation of as? sociative array subscripts during arithmetic expression evaluation, while executing builtins that can perform variable assignments, and while executing builtins that perform array dereferencing.

autocd If set, a command name that is the name of a directory is executed as if it were the argument to the cd com? mand. This option is only used by interactive shells.

cdable_vars

If set, an argument to the cd builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.

cdspell If set, minor errors in the spelling of a directory com? ponent in a cd command will be corrected. The errors checked for are transposed characters, a missing charac? ter, and one character too many. If a correction is found, the corrected filename is printed, and the com? mand proceeds. This option is only used by interactive shells.

checkhash

If set, bash checks that a command found in the hash ta? ble exists before trying to execute it. If a hashed command no longer exists, a normal path search is per? formed.

checkjobs

If set, bash lists the status of any stopped and running jobs before exiting an interactive shell. If any jobs are running, this causes the exit to be deferred until a second exit is attempted without an intervening command (see JOB CONTROL above). The shell always postpones ex? iting if any jobs are stopped.

checkwinsize

If set, bash checks the window size after each external (non-builtin) command and, if necessary, updates the values of LINES and COLUMNS. This option is enabled by default.

cmdhist If set, bash attempts to save all lines of a multiple- line command in the same history entry. This allows easy re-editing of multi-line commands. This option is enabled by default, but only has an effect if command history is enabled, as described above under HISTORY.

compat31

compat32

compat40

compat41

compat42

compat43

compat44

These control aspects of the shell's compatibility mode (see SHELL COMPATIBILITY MODE below).

complete_fullquote

If set, bash quotes all shell metacharacters in file? names and directory names when performing completion. If not set, bash removes metacharacters such as the dol? lar sign from the set of characters that will be quoted in completed filenames when these metacharacters appear in shell variable references in words to be completed. This means that dollar signs in variable names that ex?

pand to directories will not be quoted; however, any dollar signs appearing in filenames will not be quoted, either. This is active only when bash is using back? slashes to quote completed filenames. This variable is set by default, which is the default bash behavior in versions through 4.2.

direxpand

If set, bash replaces directory names with the results of word expansion when performing filename completion. This changes the contents of the readline editing buf? fer. If not set, bash attempts to preserve what the user typed.

dirspell

If set, bash attempts spelling correction on directory names during word completion if the directory name ini? tially supplied does not exist.

dotglob If set, bash includes filenames beginning with a `.' in the results of pathname expansion. The filenames ``.'' and ``..'' must always be matched explicitly, even if dotglob is set.

execfail

If set, a non-interactive shell will not exit if it can? not execute the file specified as an argument to the exec builtin command. An interactive shell does not exit if exec fails.

expand_aliases

If set, aliases are expanded as described above under ALIASES. This option is enabled by default for interac? tive shells.

extdebug

If set at shell invocation, or in a shell startup file, arrange to execute the debugger profile before the shell starts, identical to the --debugger option. If set af?

ter invocation, behavior intended for use by debuggers is enabled:

1. The -F option to the declare builtin displays the source file name and line number corresponding to each function name supplied as an argument.
2. If the command run by the DEBUG trap returns a non-zero value, the next command is skipped and not executed.
3. If the command run by the DEBUG trap returns a value of 2, and the shell is executing in a sub? routine (a shell function or a shell script exe? cuted by the . or source builtins), the shell simulates a call to return.
4. BASH_ARGC and BASH_ARGV are updated as described in their descriptions above.
5. Function tracing is enabled: command substitu? tion, shell functions, and subshells invoked with (command) inherit the DEBUG and RETURN traps.
6. Error tracing is enabled: command substitution, shell functions, and subshells invoked with (command) inherit the ERR trap.

extglob If set, the extended pattern matching features described above under Pathname Expansion are enabled.

extquote

If set, $'string' and $"string" quoting is performed within ${parameter} expansions enclosed in double quotes. This option is enabled by default.

failglob

If set, patterns which fail to match filenames during pathname expansion result in an expansion error.

force_fignore

If set, the suffixes specified by the FIGNORE shell variable cause words to be ignored when performing word

completion even if the ignored words are the only possi?
ble completions. See SHELL VARIABLES above for a de?
scription of FIGNORE. This option is enabled by de?
fault.

globasciiranges

If set, range expressions used in pattern matching bracket expressions (see Pattern Matching above) behave as if in the traditional C locale when performing com? parisons. That is, the current locale's collating se? quence is not taken into account, so b will not collate between A and B, and upper-case and lower-case ASCII characters will collate together.

globstar

If set, the pattern ** used in a pathname expansion con? text will match all files and zero or more directories and subdirectories. If the pattern is followed by a /, only directories and subdirectories match.

gnu_errfmt

If set, shell error messages are written in the standard GNU error message format.

histappend

If set, the history list is appended to the file named by the value of the HISTFILE variable when the shell ex? its, rather than overwriting the file.

histreedit

If set, and readline is being used, a user is given the opportunity to re-edit a failed history substitution.

histverify

If set, and readline is being used, the results of his? tory substitution are not immediately passed to the shell parser. Instead, the resulting line is loaded into the readline editing buffer, allowing further modi? fication.

hostcomplete

If set, and readline is being used, bash will attempt to perform hostname completion when a word containing a @ is being completed (see Completing under READLINE above). This is enabled by default.

huponexit

If set, bash will send SIGHUP to all jobs when an inter? active login shell exits.

inherit_errexit

If set, command substitution inherits the value of the errexit option, instead of unsetting it in the subshell environment. This option is enabled when posix mode is enabled.

interactive_comments

If set, allow a word beginning with # to cause that word and all remaining characters on that line to be ignored in an interactive shell (see COMMENTS above). This op? tion is enabled by default.

lastpipe

If set, and job control is not active, the shell runs the last command of a pipeline not executed in the back? ground in the current shell environment.

lithist If set, and the cmdhist option is enabled, multi-line commands are saved to the history with embedded newlines rather than using semicolon separators where possible.

localvar_inherit

If set, local variables inherit the value and attributes of a variable of the same name that exists at a previous scope before any new value is assigned. The nameref at? tribute is not inherited.

localvar_unset

If set, calling unset on local variables in previous function scopes marks them so subsequent lookups find

them unset until that function returns. This is identi?
cal to the behavior of unsetting local variables at the
current function scope.

login_shell

The shell sets this option if it is started as a login
shell (see INVOCATION above). The value may not be
changed.

mailwarn

If set, and a file that bash is checking for mail has
been accessed since the last time it was checked, the
message ``The mail in mailfile has been read'' is dis?
played.

no_empty_cmd_completion

If set, and readline is being used, bash will not at?
tempt to search the PATH for possible completions when
completion is attempted on an empty line.

nocaseglob

If set, bash matches filenames in a case-insensitive
fashion when performing pathname expansion (see Pathname
Expansion above).

nocasematch

If set, bash matches patterns in a case-insensitive
fashion when performing matching while executing case or
[[conditional commands, when performing pattern substi?
tution word expansions, or when filtering possible com?
pletions as part of programmable completion.

nullglob

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

progcomp

If set, the programmable completion facilities (see Pro?
grammable Completion above) are enabled. This option is

enabled by default.

progcomp_alias

If set, and programmable completion is enabled, bash treats a command name that doesn't have any completions as a possible alias and attempts alias expansion. If it has an alias, bash attempts programmable completion us? ing the command word resulting from the expanded alias.

promptvars

If set, prompt strings undergo parameter expansion, com? mand substitution, arithmetic expansion, and quote re? moval after being expanded as described in PROMPTING above. This option is enabled by default.

restricted_shell

The shell sets this option if it is started in re? stricted mode (see RESTRICTED SHELL below). The value may not be changed. This is not reset when the startup files are executed, allowing the startup files to dis? cover whether or not a shell is restricted.

shift_verbose

If set, the shift builtin prints an error message when the shift count exceeds the number of positional parame? ters.

sourcepath

If set, the source (.) builtin uses the value of PATH to find the directory containing the file supplied as an argument. This option is enabled by default.

syslog_history

If set, command history is logged to syslog.

xpg_echo

If set, the echo builtin expands backslash-escape se? quences by default.

suspend [-f]

Suspend the execution of this shell until it receives a SIGCONT

signal. A login shell cannot be suspended; the -f option can be used to override this and force the suspension. The return sta? tus is 0 unless the shell is a login shell and -f is not sup? plied, or if job control is not enabled.

test expr

[expr]

Return a status of 0 (true) or 1 (false) depending on the evalu? ation of the conditional expression expr. Each operator and op? erand must be a separate argument. Expressions are composed of the primaries described above under CONDITIONAL EXPRESSIONS. test does not accept any options, nor does it accept and ignore an argument of -- as signifying the end of options.

Expressions may be combined using the following operators, listed in decreasing order of precedence. The evaluation de? pends on the number of arguments; see below. Operator prece? dence is used when there are five or more arguments.

! expr True if expr is false.

(expr)

Returns the value of expr. This may be used to override the normal precedence of operators.

expr1 -a expr2

True if both expr1 and expr2 are true.

expr1 -o expr2

True if either expr1 or expr2 is true.

test and [evaluate conditional expressions using a set of rules based on the number of arguments.

0 arguments

The expression is false.

1 argument

The expression is true if and only if the argument is not null.

2 arguments

If the first argument is !, the expression is true if and

only if the second argument is null. If the first argu?
ment is one of the unary conditional operators listed
above under CONDITIONAL EXPRESSIONS, the expression is
true if the unary test is true. If the first argument is
not a valid unary conditional operator, the expression is
false.

3 arguments

The following conditions are applied in the order listed.
If the second argument is one of the binary conditional
operators listed above under CONDITIONAL EXPRESSIONS, the
result of the expression is the result of the binary test
using the first and third arguments as operands. The -a
and -o operators are considered binary operators when
there are three arguments. If the first argument is !,
the value is the negation of the two-argument test using
the second and third arguments. If the first argument is
exactly (and the third argument is exactly), the result
is the one-argument test of the second argument. Other?
wise, the expression is false.

4 arguments

If the first argument is !, the result is the negation of
the three-argument expression composed of the remaining
arguments. Otherwise, the expression is parsed and eval?
uated according to precedence using the rules listed
above.

5 or more arguments

The expression is parsed and evaluated according to
precedence using the rules listed above.

When used with test or [, the < and > operators sort lexico?
graphically using ASCII ordering.

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

trap [-lp] [[arg] sigspec ...]

The command arg is to be read and executed when the shell re?
ceives signal(s) sigspec. If arg is absent (and there is a sin?
gle sigspec) or -, each specified signal is reset to its origi?
nal disposition (the value it had upon entrance to the shell).
If arg is the null string the signal specified by each sigspec
is ignored by the shell and by the commands it invokes. If arg
is not present and -p has been supplied, then the trap commands
associated with each sigspec are displayed. If no arguments are
supplied or if only -p is given, trap prints the list of com?
mands associated with each signal. The -l option causes the
shell to print a list of signal names and their corresponding
numbers. Each sigspec is either a signal name defined in <sig?
nal.h>, or a signal number. Signal names are case insensitive
and the SIG prefix is optional.
If a sigspec is EXIT (0) the command arg is executed on exit
from the shell. If a sigspec is DEBUG, the command arg is exe?
cuted before every simple command, for command, case command,
select command, every arithmetic for command, and before the
first command executes in a shell function (see SHELL GRAMMAR
above). Refer to the description of the extdebug option to the
shopt builtin for details of its effect on the DEBUG trap. If a
sigspec is RETURN, the command arg is executed each time a shell
function or a script executed with the . or source builtins fin?
ishes executing.
If a sigspec is ERR, the command arg is executed whenever a
pipeline (which may consist of a single simple command), a list,
or a compound command returns a non-zero exit status, subject to
the following conditions. The ERR trap is not executed if the
failed command is part of the command list immediately following
a while or until keyword, part of the test in an if statement,
part of a command executed in a && or || list except the command
following the final && or ||, any command in a pipeline but the
last, or if the command's return value is being inverted using

!. These are the same conditions obeyed by the errexit (-e) op?
tion.

Signals ignored upon entry to the shell cannot be trapped, reset or listed. Trapped signals that are not being ignored are reset to their original values in a subshell or subshell environment when one is created. The return status is false if any sigspec is invalid; otherwise trap returns true.

type [-aftpP] name [name ...]

With no options, indicate how each name would be interpreted if used as a command name. If the -t option is used, type prints a string which is one of alias, keyword, function, builtin, or file if name is an alias, shell reserved word, function, builtin, or disk file, respectively. If the name is not found, then nothing is printed, and an exit status of false is re?
turned. If the -p option 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 -t name'' would not re?
turn file. The -P option forces a PATH search for each name, even if ``type -t name'' would not return file. If a command is hashed, -p and -P print the hashed value, which is not necessar?
ily the file that appears first in PATH. If the -a option is used, type prints all of the places that contain an executable named name. This includes aliases and functions, if and only if the -p option is not also used. The table of hashed commands is not consulted when using -a. The -f option suppresses shell function lookup, as with the command builtin. type returns true if all of the arguments are found, false if any are not found.

ulimit [-HS] -a

ulimit [-HS] [-bcdefiklmnpqrstuvxPRT [limit]]

Provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The -H and -S options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased by a

non-root user once it is set; a soft limit may be increased up to the value of the hard limit. If neither -H nor -S is speci? fied, both the soft and hard limits are set. The value of limit can be a number in the unit specified for the resource or one of the special values hard, soft, or unlimited, which stand for the current hard limit, the current soft limit, and no limit, re? spectively. 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, if appropriate, are printed before the value. Other op? tions are interpreted as follows:

-a All current limits are reported; no limits are set

-b The maximum socket buffer size

-c The maximum size of core files created

-d The maximum size of a process's data segment

-e The maximum scheduling priority ("nice")

-f The maximum size of files written by the shell and its children

-i The maximum number of pending signals

-k The maximum number of kqueues that may be allocated

-l The maximum size that may be locked into memory

-m The maximum resident set size (many systems do not honor this limit)

-n The maximum number of open file descriptors (most systems do not allow this value to be set)

-p The pipe size in 512-byte blocks (this may not be set)

-q The maximum number of bytes in POSIX message queues

-r The maximum real-time scheduling priority

-s The maximum stack size

-t The maximum amount of cpu time in seconds

-u The maximum number of processes available to a single user

-v The maximum amount of virtual memory available to the

shell and, on some systems, to its children

-x The maximum number of file locks

-P The maximum number of pseudoterminals

-R The maximum time a real-time process can run before blocking, in microseconds

-T The maximum number of threads

If limit is given, and the -a option is not used, limit is the new value of the specified resource. If no option is given, then -f is assumed. Values are in 1024-byte increments, except for -t, which is in seconds; -R, which is in microseconds; -p, which is in units of 512-byte blocks; -P, -T, -b, -k, -n, and -u, which are unscaled values; and, when in posix mode, -c and -f, which are in 512-byte increments. The return status is 0 unless an invalid option or argument is supplied, or an error occurs while setting a new limit. In POSIX Mode 512-byte blocks are used for the `-c' and `-f' options.

umask [-p] [-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, the current value of the mask is printed. The -S option causes the mask to be printed in sym? bolic form; the default output is an octal number. If the -p option is supplied, and mode is omitted, the output is in a form that may be reused as input. The return status is 0 if the mode was successfully changed or if no mode argument was supplied, and false otherwise.

unalias [-a] [name ...]

Remove each name from the list of defined aliases. If -a is supplied, all alias definitions are removed. The return value is true unless a supplied name is not a defined alias.

unset [-fv] [-n] [name ...]

For each name, remove the corresponding variable or function.

If the -v option is given, each name refers to a shell variable, and that variable is removed. Read-only variables may not be unset. If -f is specified, each name refers to a shell func? tion, and the function definition is removed. If the -n option is supplied, and name is a variable with the nameref attribute, name will be unset rather than the variable it references. -n has no effect if the -f option is supplied. If no options are supplied, each name refers to a variable; if there is no vari? able by that name, a function with that name, if any, is unset. Each unset variable or function is removed from the environment passed to subsequent commands. If any of BASH_ALIASES, BASH_ARGV0, BASH_CMDS, BASH_COMMAND, BASH_SUBSHELL, BASHPID, COMP_WORDBREAKS, DIRSTACK, EPOCHREALTIME, EPOCHSECONDS, FUNC? NAME, GROUPS, HISTCMD, LINENO, RANDOM, SECONDS, or SRANDOM are unset, they lose their special properties, even if they are sub? sequently reset. The exit status is true unless a name is read? only.

wait [-fn] [-p varname] [id ...]

Wait for each specified child process and return its termination status. Each id 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 id is not given, wait waits for all running background jobs and the last-executed process substitution, if its process id is the same as $!, and the return status is zero. If the -n option is supplied, wait waits for a single job from the list of ids or, if no ids are supplied, any job, to complete and returns its exit status. If none of the supplied arguments is a child of the shell, or if no arguments are supplied and the shell has no unwaited-for children, the exit status is 127. If the -p option is supplied, the process or job identifier of the job for which the exit status is returned is assigned to the variable varname named by the option argument. The variable will be unset initially, before any assignment. This is useful

only when the -n option is supplied. Supplying the -f option, when job control is enabled, forces wait to wait for id to ter? minate before returning its status, instead of returning when it changes status. If id specifies a non-existent process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

SHELL COMPATIBILITY MODE

Bash-4.0 introduced the concept of a `shell compatibility level', spec? ified as a set of options to the shopt builtin compat31, compat32, com? pat40, compat41, and so on). There is only one current compatibility level -- each option is mutually exclusive. The compatibility level is intended to allow users to select behavior from previous versions that is incompatible with newer versions while they migrate scripts to use current features and behavior. It's intended to be a temporary solu? tion.

This section does not mention behavior that is standard for a particu? lar version (e.g., setting compat32 means that quoting the rhs of the regexp matching operator quotes special regexp characters in the word, which is default behavior in bash-3.2 and above).

If a user enables, say, compat32, it may affect the behavior of other compatibility levels up to and including the current compatibility level. The idea is that each compatibility level controls behavior that changed in that version of bash, but that behavior may have been present in earlier versions. For instance, the change to use locale-based comparisons with the [[command came in bash-4.1, and earlier versions used ASCII-based comparisons, so enabling compat32 will enable ASCII-based comparisons as well. That granularity may not be suffi? cient for all uses, and as a result users should employ compatibility levels carefully. Read the documentation for a particular feature to find out the current behavior.

Bash-4.3 introduced a new shell variable: BASH_COMPAT. The value as? signed to this variable (a decimal version number like 4.2, or an inte? ger corresponding to the compatNN option, like 42) determines the com?

patibility level.

Starting with bash-4.4, Bash has begun deprecating older compatibility levels. Eventually, the options will be removed in favor of BASH_COM? PAT.

Bash-5.0 is the final version for which there will be an individual shopt option for the previous version. Users should use BASH_COMPAT on bash-5.0 and later versions.

The following table describes the behavior changes controlled by each compatibility level setting. The compatNN tag is used as shorthand for setting the compatibility level to NN using one of the following mecha? nisms. For versions prior to bash-5.0, the compatibility level may be set using the corresponding compatNN shopt option. For bash-4.3 and later versions, the BASH_COMPAT variable is preferred, and it is re? quired for bash-5.1 and later versions.

compat31

? quoting the rhs of the [[command's regexp matching oper? ator (=~) has no special effect

compat32

? interrupting a command list such as "a ; b ; c" causes the execution of the next command in the list (in bash-4.0 and later versions, the shell acts as if it re? ceived the interrupt, so interrupting one command in a list aborts the execution of the entire list)

compat40

? the < and > operators to the [[command do not consider the current locale when comparing strings; they use ASCII ordering. Bash versions prior to bash-4.1 use ASCII col? lation and strcmp(3); bash-4.1 and later use the current locale's collation sequence and strcoll(3).

compat41

? in posix mode, time may be followed by options and still be recognized as a reserved word (this is POSIX interpre? tation 267)

? in posix mode, the parser requires that an even number of single quotes occur in the word portion of a double-quoted parameter expansion and treats them specially, so that characters within the single quotes are considered quoted (this is POSIX interpretation 221)

compat42

? the replacement string in double-quoted pattern substitu? tion does not undergo quote removal, as it does in ver? sions after bash-4.2

? in posix mode, single quotes are considered special when expanding the word portion of a double-quoted parameter expansion and can be used to quote a closing brace or other special character (this is part of POSIX interpre? tation 221); in later versions, single quotes are not special within double-quoted word expansions

compat43

? the shell does not print a warning message if an attempt is made to use a quoted compound assignment as an argu? ment to declare (declare -a foo='(1 2)'). Later versions warn that this usage is deprecated

? word expansion errors are considered non-fatal errors that cause the current command to fail, even in posix mode (the default behavior is to make them fatal errors that cause the shell to exit)

? when executing a shell function, the loop state (while/until/etc.) is not reset, so break or continue in that function will break or continue loops in the calling context. Bash-4.4 and later reset the loop state to pre? vent this

compat44

? the shell sets up the values used by BASH_ARGV and BASH_ARGC so they can expand to the shell's positional parameters even if extended debugging mode is not enabled

? a subshell inherits loops from its parent context, so break or continue will cause the subshell to exit. Bash-5.0 and later reset the loop state to prevent the exit

? variable assignments preceding builtins like export and readonly that set attributes continue to affect variables with the same name in the calling environment even if the shell is not in posix mode

compat50

? Bash-5.1 changed the way $RANDOM is generated to intro? duce slightly more randomness. If the shell compatibility level is set to 50 or lower, it reverts to the method from bash-5.0 and previous versions, so seeding the ran? dom number generator by assigning a value to RANDOM will produce the same sequence as in bash-5.0

? If the command hash table is empty, bash versions prior to bash-5.1 printed an informational message to that ef? fect, even when producing output that can be reused as input. Bash-5.1 suppresses that message when the -l op? tion is supplied.

RESTRICTED SHELL

If bash is started with the name rbash, or the -r option is supplied at invocation, the shell becomes restricted. A restricted shell is used to set up an environment more controlled than the standard shell. It behaves identically to bash with the exception that the following are disallowed or not performed:

? changing directories with cd

? setting or unsetting the values of SHELL, PATH, HISTFILE, ENV, or BASH_ENV

? specifying command names containing /

? specifying a filename containing a / as an argument to the . builtin command

? specifying a filename containing a slash as an argument to the

history builtin command

? specifying a filename containing a slash as an argument to the -p option to the hash builtin command

? importing function definitions from the shell environment at startup

? parsing the value of SHELLOPTS from the shell environment at startup

? redirecting output using the >, >|, <>, >&, &>, and >> redirect? ion operators

? using the exec builtin command to replace the shell with another command

? adding or deleting builtin commands with the -f and -d options to the enable builtin command

? using the enable builtin command to enable disabled shell builtins

? specifying the -p option to the command builtin command

? turning off restricted mode with set +r or set +o restricted.

These restrictions are enforced after any startup files are read.

When a command that is found to be a shell script is executed (see COM? MAND EXECUTION above), rbash turns off any restrictions in the shell spawned to execute the script.

SEE ALSO

Bash Reference Manual, Brian Fox and Chet Ramey

The Gnu Readline Library, Brian Fox and Chet Ramey

The Gnu History Library, Brian Fox and Chet Ramey

Portable Operating System Interface (POSIX) Part 2: Shell and Utili? ties, IEEE --

http://pubs.opengroup.org/onlinepubs/9699919799/

http://tiswww.case.edu/~chet/bash/POSIX -- a description of posix mode

sh(1), ksh(1), csh(1)

emacs(1), vi(1)

readline(3)

FILES

/bin/bash

The bash executable

/etc/profile

The systemwide initialization file, executed for login shells

/etc/bash.bash_logout

The systemwide login shell cleanup file, executed when a login shell exits

~/.bash_profile

The personal initialization file, executed for login shells

~/.bashrc

The individual per-interactive-shell startup file

~/.bash_logout

The individual login shell cleanup file, executed when a login shell exits

~/.inputrc

Individual readline initialization file

AUTHORS

Brian Fox, Free Software Foundation

bfox@gnu.org

Chet Ramey, Case Western Reserve University

chet.ramey@case.edu

BUG REPORTS

If you find a bug in bash, you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of bash. The latest version is always available from ftp://ftp.gnu.org/pub/gnu/bash/.

Once you have determined that a bug actually exists, use the bashbug command to submit a bug report. If you have a fix, you are encouraged to mail that as well! Suggestions and `philosophical' bug reports may be mailed to bug-bash@gnu.org or posted to the Usenet newsgroup gnu.bash.bug.

ALL bug reports should include:

The version number of bash

The hardware and operating system

The compiler used to compile

A description of the bug behaviour

A short script or `recipe' which exercises the bug

bashbug inserts the first three items automatically into the template it provides for filing a bug report.

Comments and bug reports concerning this manual page should be directed to chet.ramey@case.edu.

BUGS

It's too big and too slow.

There are some subtle differences between bash and traditional versions of sh, mostly because of the POSIX specification.

Aliases are confusing in some uses.

Shell builtin commands and functions are not stoppable/restartable.

Compound commands and command sequences of the form `a ; b ; c' are not handled gracefully when process suspension is attempted. When a process is stopped, the shell immediately executes the next command in the sequence. It suffices to place the sequence of commands between parentheses to force it into a subshell, which may be stopped as a unit.

Array variables may not (yet) be exported.

There may be only one active coprocess at a time.