Bash Reference Manual

Reference Documentation for Bash
Edition 2.0, for `bash` Version 2.01.
19 May 1997

Chet Ramey, Case Western Reserve University
Brian Fox, Free Software Foundation

Copyright © 1991, 1993, 1996 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

1 Introduction

1.1 What is Bash?

Bash is the shell, or command language interpreter, that will appear in the GNU operating system. The name is an acronym for the '`Bourne-Again SHell`', a pun on Steve Bourne, the author of the direct ancestor of the current Unix shell `/bin/sh`, which appeared in the Seventh Edition Bell Labs Research version of Unix.

Bash is an `sh`-compatible shell that incorporates useful features from the Korn shell `ksh` and the C shell `csh`. It is ultimately intended to be a conformant implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2). It offers functional improvements over `sh` for both interactive and programming use.

While the GNU operating system will include a version of `csh`, Bash will be the default shell. Like other GNU software, Bash is quite portable. It currently runs on nearly every version of Unix and a few other operating systems − independently-supported ports exist for OS/2 and Windows NT.

1.2 What is a shell?

At its base, a shell is simply a macro processor that executes commands. A Unix shell is both a command interpreter, which provides the user interface to the rich set of Unix utilities, and a programming language, allowing these utilitites to be combined. The shell reads commands either from a terminal or a file. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories like '`/bin`', allowing users or groups to establish custom environments.

A shell allows execution of Unix commands, both synchronously and asynchronously. The *redirection* constructs permit fine-grained control of the input and output of those commands, and the shell allows control over the contents of their environment. Unix shells also provide a small set of built-in commands (*builtins*) implementing functionality impossible (e.g., `cd`, `break`, `continue`, and `exec`), or inconvenient (`history`, `getopts`, `kill`, or `pwd`, for example) to obtain via separate utilities. Shells may be used interactively or non-interactively: they accept input typed from the keyboard or from a file. All of the shell builtins are described in subsequent sections.

While executing commands is essential, most of the power (and complexity) of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions.

Shells have begun offering features geared specifically for interactive use rather than to augment the programming language. These interactive features include job control, command line editing, history and aliases. Each of these features is described in this manual.

2 Definitions

These definitions are used throughout the remainder of this manual.

`POSIX`
: A family of open system standards based on Unix. Bash is concerned with POSIX 1003.2, the Shell and Tools Standard.

`blank`
: A space or tab character.

`builtin`
: A command that is implemented internally by the shell itself, rather than by an executable program somewhere in the file system.

`control operator`
: A `word` that performs a control function. It is a `newline` or one of the following: '`||`', '`&&`', '`&`', '`;`', '`;;`', '`|`', '`(`', or '`)`'.

`exit status`
: The value returned by a command to its caller.

`field`
: A unit of text that is the result of one of the shell expansions. After expansion, when executing a command, the resulting fields are used as the command name and arguments.

`filename`
: A string of characters used to identify a file.

`job`
: A set of processes comprising a pipeline, and any processes descended from it, that are all in the same process group.

`job control`
: A mechanism by which users can selectively stop (suspend) and restart (resume) execution of processes.

`metacharacter`
: A character that, when unquoted, separates words. A metacharacter is a `blank` or one of the following characters: '`|`', '`&`', '`;`', '`(`', '`)`', '`<`', or '`>`'.

`name`
: A `word` consisting solely of letters, numbers, and underscores, and beginning with a letter or underscore. `Name`s are used as shell variable and function names. Also referred to as an `identifier`.

`operator`
: A `control operator` or a `redirection operator`. See Section 3.11 [Redirections], page 20, for a list of redirection operators.

`process group`
: A collection of related processes each having the same process group ID.

`process group ID`
: A unique identifer that represents a `process group` during its lifetime.

`reserved word`
: A `word` that has a special meaning to the shell. Most reserved words introduce shell flow control constructs, such as `for` and `while`.

`return status`
: A synonym for `exit status`.

`signal`
: A mechanism by which a process may be notified by the kernal of an event occurring in the system.

`special builtin`
: A shell builtin command that has been classified as special by the POSIX.2 standard.

`token`
: A sequence of characters considered a single unit by the shell. It is either a `word` or an `operator`.

`word`
: A `token` that is not an `operator`.

3 Basic Shell Features

Bash is an acronym for '`Bourne-Again SHell`'. The Bourne shell is the traditional Unix shell originally written by Stephen Bourne. All of the Bourne shell builtin commands are available in Bash, and the rules for evaluation and quoting are taken from the POSIX 1003.2 specification for the 'standard' Unix shell.

This chapter briefly summarizes the shell's "building blocks": commands, control structures, shell functions, shell *parameters*, shell expansions, *redirections*, which are a way to direct input and output from and to named files, and how the shell executes commands.

3.1 Shell Syntax

3.1.1 Shell Operation

The following is a brief description of the shell's operation when it reads and executes a command. Basically, the shell does the following:

1. Reads its input from a file (see Section 3.13 [Shell Scripts], page 26), from a string supplied as an argument to the '`-c`' invocation option (see Section 7.1 [Invoking Bash], page 49), or from the user's terminal.
2. Breaks the input into words and operators, obeying the quoting rules described in Section 3.1.2 [Quoting], page 6. Tokens are separated by `metacharacters`. Alias expansion is performed by this step (see Section 6.4 [Aliases], page 47).
3. Parses the tokens into simple and compound commands.
4. Performs the various shell expansions (see Section 3.10 [Shell Expansions], page 14), breaking the expanded tokens into lists of filenames (see Section 3.10.5 [Filename Expansion], page 19) and commands and arguments.
5. Performs any necessary redirections (see Section 3.11 [Redirections], page 20) and removes the redirection operators and their operands from the argument list.
6. Executes the command (see Section 3.12 [Executing Commands], page 24).
7. Optionally waits for the command to complete and collects its exit status.

3.1.2 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 shell `metacharacters` (see Chapter 2 [Definitions], page 3) has special meaning to the shell and must be quoted if they are to represent themselves. There are three quoting mechanisms: the *escape character*, single quotes, and double quotes.

3.1.2.1 Escape Character

A non-quoted backslash ‘\’ is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of `newline`. If a `\newline` pair appears, and the backslash is not quoted, the `\newline` is treated as a line continuation (that is, it is effectively ignored).

3.1.2.2 Single Quotes

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.

3.1.2.3 Double Quotes

Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of ‘$’, ‘`’, and ‘\’. The characters ‘$’ and ‘`’ retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: ‘$’, ‘`’, ‘"’, ‘\’, or `newline`. A double quote may be quoted within double quotes by preceding it with a backslash.

The special parameters ‘*’ and ‘@’ have special meaning when in double quotes (see Section 3.10.1 [Shell Parameter Expansion], page 15).

3.1.2.4 ANSI-C Quoting

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

`\a` alert (bell)

`\b` backspace

`\e` an escape character (not ANSI C)

`\f` form feed

`\n` newline

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\\` backslash

`\nnn` the character whose `ASCII` code is *nnn* in octal

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

3.1.2.5 Locale-Specific Translation

A double-quoted string preceded by a dollar sign ('`$`') will cause the string to be translated according to the current locale. If the current locale is `C` or `POSIX`, the dollar sign is ignored. If the string is translated and replaced, the replacement is double-quoted.

3.1.3 Comments

In a non-interactive shell, or an interactive shell in which the `interactive_comments` option to the `shopt` builtin is enabled (see Section 7.4 [Bash Builtins], page 53), 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 interactive shells.

3.2 Simple Commands

A simple command is the kind of command you'll encounter most often. It's just a sequence of words separated by `blank`s, terminated by one of the shell control operators (see Chapter 2 [Definitions], page 3). The first word generally specifies a command to be executed.

The return status (see Section 3.12.3 [Exit Status], page 25) of a simple command is its exit status as provided by the POSIX.1 `waitpid` function, or 128+*n* if the command was terminated by signal *n*.

3.3 Pipelines

A `pipeline` is a sequence of simple commands separated by '`|`'.

The format for a pipeline is

`[time [-p]] [!]` *command1* `[|` *command2* ...]

The output of each command in the pipeline is connected to the input of the next command. That is, each command reads the previous command's output.

The reserved word `time` causes timing statistics to be printed for the pipeline once it finishes. The '`-p`' option changes the output format to that specified by POSIX. The `TIMEFORMAT` variable may be set to a format string that specifies how the timing information should be displayed. See Section 7.7 [Bash Variables], page 65, for a description of the available formats.

Each command in a pipeline is executed in its own subshell. The exit status of a pipeline is the exit status of the last command in the pipeline. If the reserved word '`!`' precedes the pipeline, the exit status is the logical NOT of the exit status of the last command.

3.4 Lists of Commands

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

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

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

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

command `&&` *command2*

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

An OR list has the form

command `||` *command2*

command2 is executed if and only if *command* returns a non-zero exit status.

The return status of AND and OR lists is the exit status of the last command executed in the list.

3.5 Looping Constructs

Note that wherever you see a '`;`' in the description of a command's syntax, it may be replaced indiscriminately with one or more newlines.

Bash supports the following looping constructs.

`until` The syntax of the `until` command is:

`until` *test-commands*`; do` *consequent-commands*`; done`

Execute *consequent-commands* as long as the final command in *test-commands* has an exit status which is not zero.

`while` The syntax of the `while` command is:

`while` *test-commands*`; do` *consequent-commands*`; done`

Execute *consequent-commands* as long as the final command in *test-commands* has an exit status of zero.

`for` The syntax of the `for` command is:

`for` *name* `[in` *words* `...]; do` *commands*`; done`

Execute *commands* for each member in *words*, with *name* bound to the current member. If '`in` *words*' is not present, '`in "$@"`' is assumed.

The `break` and `continue` builtins (see Section 4.1 [Bourne Shell Builtins], page 29) may be used to control loop execution.

3.6 Conditional Constructs

`if` The syntax of the `if` command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

Execute *consequent-commands* only if the final command in *test-commands* has an exit status of zero. Otherwise, each `elif` list is executed in turn, and if its exit status is zero, the corresponding *more-consequents* is executed and the command completes. If '`else` *alternate-consequents*' is present, and the final command in the final `if` or `elif` clause has a non-zero exit status, then execute *alternate-consequents*.

`case` The syntax of the `case` command is:

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

Selectively execute *commands* based upon *word* matching *pattern*. The '`|`' is used to separate multiple patterns.

Here is an example using `case` in a script that could be used to describe one interesting feature of an animal:

```
echo -n "Enter the name of an animal: "
read ANIMAL
echo -n "The $ANIMAL has "
case $ANIMAL in
  horse | dog | cat) echo -n "four";;
  man | kangaroo ) echo -n "two";;
  *) echo -n "an unknown number of";;
```

```
esac
echo " legs."
```

`((...))`

`(( expression ))`

The *expression* is evaluated according to the rules described below (see Section 7.8.1 [Arithmetic Evaluation], page 69). 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"`

The `select` construct, which allows users to choose from a list of items presented as a menu, is also available. See Section 6.1 [Korn Shell Constructs], page 45, for a full description of `select`.

3.7 Grouping Commands

Bash provides two ways to group a list of commands to be executed as a unit. When commands are grouped, redirections may be applied to the entire command list. For example, the output of all the commands in the list may be redirected to a single stream.

`()`

`( list )`

Placing a list of commands between parentheses causes a subshell to be created, and each of the commands to be executed in that subshell. Since the *list* is executed in a subshell, variable assignments do not remain in effect after the subshell completes.

`{}`

`{ list; }`

Placing a list of commands between curly braces causes the list to be executed in the current shell context. No subshell is created. The semicolon following *list* is required.

In addition to the creation of a subshell, there is a subtle difference between these two constructs due to historical reasons. The braces are `reserved words`, so they must be separated from the *list* by `blanks`. The parentheses are `operators`, and are recognized as separate tokens by the shell even if they are not separated from the *list* by whitespace.

The exit status of both of these constructs is the exit status of *list*.

3.8 Shell Functions

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a "regular" command. Shell functions are executed in the current shell context; no new process is created to interpret them.

Functions are declared using this syntax:

```
[ function ] name () { command-list; }
```

This defines a shell function named *name*. The reserved word `function` is optional. The *body* of the function is the *command-list* between `{` and `}`. This list is executed whenever *name* is specified as the name of a command. The exit status of a function is the exit status of the last command executed in the body.

When a function is executed, the arguments to the function become the positional parameters during its execution (see Section 3.9.1 [Positional Parameters], page 13). The special parameter '`#`' that expands to the number of positional parameters is updated to reflect the change. Positional parameter `0` is unchanged.

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

Variables local to the function may be declared with the `local` builtin. These variables are visible only to the function and the commands it invokes.

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

3.9 Shell Parameters

A *parameter* is an entity that stores values. It can be a `name`, a number, or one of the special characters listed below. For the shell's purposes, a *variable* is a parameter denoted by a `name`.

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the `unset` builtin command.

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 *value*s undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (detailed below). If the variable has its '`-i`' attribute set (see the description of the `declare` builtin in Section 7.4 [Bash Builtins], page 53), then *value* is subject to arithmetic expansion even if the `$((...))` syntax does not appear (see Section 7.8.2 [Arithmetic Expansion], page 70). Word splitting is not performed, with the exception of `"$@"` as explained below. Filename expansion is not performed.

3.9.1 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 Section 3.8 [Shell Functions], page 12).

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.

3.9.2 Special Parameters

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

`*` Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the `IFS` special variable. That is, `"$*"` is equivalent to `"$1c$2c..."`, where *c* is the first character of the value of the `IFS` variable. If `IFS` is 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. When the expansion occurs within double quotes, each parameter expands as a separate word. That is, `"$@"` is equivalent to `"$1" "$2" ...`. When there are no positional parameters, `"$@"` and `$@` expand to nothing (i.e., they are removed).

`#` Expands to the number of positional parameters in decimal.

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

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

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

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

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

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

3.10 Shell Expansions

Expansion is performed on the command line after it has been split into `token`s. There are seven kinds of expansion performed:

- brace expansion
- tilde expansion
- parameter and variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion

Brace expansion, tilde expansion, and arithmetic expansion are described in other sections. For brace expansion, see Section 5.1 [Brace Expansion], page 41; for tilde expansion, see Section 5.2 [Tilde Expansion], page 42; and for arithmetic expansion, see Section 7.8.2 [Arithmetic Expansion], page 70.

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

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

Only brace expansion, word splitting, and filename expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of `"$@"` (see Section 3.9.2 [Special Parameters], page 13) and `"${`*name*`[@]}"` (see Section 7.9 [Arrays], page 71).

After all expansions, `quote removal` (see Section 3.10.6 [Quote Removal], page 20) is performed.

3.10.1 Shell 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.

The basic form of parameter expansion is ${*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 that is not to be interpreted as part of its name.

If the first character of *parameter* is an exclamation point, a level of variable indirection is introduced. Bash uses the value of the variable formed from the rest of *parameter* as the name of the variable; this variable is then expanded and that value is used in the rest of the substitution, rather than the value of *parameter* itself. This is known as `indirect expansion`.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. When not performing substring expansion, Bash tests for a parameter that is unset or null; omitting the colon results in a test only for a parameter that is unset.

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

`${`*parameter*`:=`*word*`}`
: 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*`}`
: 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*`}`
: If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.

`${`*parameter*`:`*offset*`}`
`${`*parameter*`:`*offset*`:`*length*`}`
: Expands to up to *length* characters of *parameter*, starting at *offset*. If *length* is omitted, expands to the substring of *parameter*, starting at the character specified by *offset*. *length* and *offset* are arithmetic expressions (see Section 7.8.1 [Arithmetic Evaluation], page 69). This is referred to as Substring Expansion.

  length must evaluate to a number greater than or equal to zero. If *offset* evaluates to a number less than zero, the value is used as an offset from the end of the value of *parameter*. If *parameter* is '`@`', the result is *length* positional parameters beginning at *offset*. If *parameter* is an array name indexed by '`@`' or '`*`', the result is the *length* members of the array beginning with ${*parameter*[*offset*]}. Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1.

`${#`*parameter*`}`
: The length in characters of the value of *parameter* is substituted. If *parameter* is '`*`' or '`@`', the length substituted is the number of positional parameters. If *parameter* is an array name subscripted by '`*`' or '`@`', the length substituted is the number of elements in the array.

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

The *word* is expanded to produce a pattern just as in filename expansion (see Section 3.10.5 [Filename Expansion], page 19). If the pattern matches the beginning of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern (the '`#`' case) or the longest matching pattern (the '`##`' case) deleted. If *parameter* is '`@`' or '`*`', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

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

The *word* is expanded to produce a pattern just as in filename expansion. If the pattern matches a trailing portion of the value of *parameter*, then the expansion is the value of *parameter* with the shortest matching pattern (the '`%`' case) or the longest matching pattern (the '`%%`' case) deleted. If *parameter* is '`@`' or '`*`', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

`${`*parameter*`/`*pattern*`/`*string*`}`
`${`*parameter*`//`*pattern*`/`*string*`}`

The *pattern* is expanded to produce a pattern just as in filename expansion. *Parameter* is expanded and the longest match of *pattern* against its value is replaced with *string*. In the first form, only the first match is replaced. The second form causes all matches of *pattern* to be replaced with *string*. If *pattern* begins with '`#`', it must match at the beginning of *string*. If *pattern* begins with '`%`', it must match at the end of *string*. If *string* is null, matches of *pattern* are deleted and the `/` following *pattern* may be omitted. If *parameter* is '`@`' or '`*`', the substitution operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the substitution operation is applied to each member of the array in turn, and the expansion is the resultant list.

3.10.2 Command Substitution

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

`$(`*command*`)`

or

`command`

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

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

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

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

3.10.3 Process Substitution

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

`<(`*list*`)`

or

`>(`*list*`)`

The process *list* is run with its input or output connected to a FIFO or some file in '`/dev/fd`'. The name of this file is passed as an argument to the current command as the result of the expansion. If the `>(`*list*`)` form is used, writing to the file will provide input for *list*. If the `<(`*list*`)` form is used, the file passed as an argument should be read to obtain the output of *list*.

On systems that support it, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

3.10.4 Word Splitting

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

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

Explicit null arguments (`""` or `''`) are retained. Unquoted implicit null arguments, resulting from the expansion of *parameter*s that have no values, are removed. If a parameter with no value is expanded within double quotes, a null argument results and is retained.

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

3.10.5 Filename Expansion

After word splitting, unless the '`-f`' option has been set (see Section 7.5 [The Set Builtin], page 60), Bash scans each word for the characters '`*`', '`?`', and '`[`'. If one of these characters appears, then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of file names matching the pattern. If no matching file names are found, and the shell option `nullglob` is disabled, the word is left unchanged. If the option is set, and no matches are found, the word is removed. When a pattern is used for filename generation, the character '`.`' at the start of a filename or immediately following a slash must be matched explicitly, unless the shell option `dotglob` is set. The slash character must always be matched explicitly. In other cases, the '`.`' character is not treated specially. See the description of `shopt` in Section 7.4 [Bash Builtins], page 53, for a description of the `nullglob` and `dotglob` options.

The `GLOBIGNORE` shell variable may be used to restrict the set of filenames matching a *pattern*. If `GLOBIGNORE` is set, each matching filename that also matches one of the patterns in `GLOBIGNORE` is removed from the list of matches. The filenames '`.`' and '`..`' are always ignored, even when `GLOBIGNORE`. is set. However, setting `GLOBIGNORE` 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 special pattern characters have the following meanings:

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

`?` Matches any single character.

`[...]` Matches any one of the enclosed characters. A pair of characters separated by a minus sign denotes a *range*; any character lexically between those two characters, inclusive, is matched. If the first character following the '[' is a '!' or a '^' then any character not enclosed is matched. A '−' 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.

3.10.6 Quote Removal

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

3.11 Redirections

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

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

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

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

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

```
ls 2>&1 > dirlist
```

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

3.11.1 Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor `n`, or the standard input (file descriptor 0) if `n` is not specified.

The general format for redirecting input is:

```
[n]<word
```

3.11.2 Redirecting Output

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

The general format for redirecting output is:

```
[n]>[|]word
```

If the redirection operator is '`>`', and the '`-C`' option to the `set` builtin has been enabled, the redirection will fail if the filename whose name results from the expansion of *word* exists. If the redirection operator is '`>|`', then the value of the '`-C`' option to the `set` builtin command is not tested, and the redirection is attempted even if the file named by *word* exists.

3.11.3 Appending Redirected Output

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

The general format for appending output is:

`[n]>>`*word*

3.11.4 Redirecting Standard Output and Standard Error

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

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

`&>`*word*

and

`>&`*word*

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

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

3.11.5 Here Documents

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

The format of here-documents is as follows:

`<<[-]`*word*

here-document
delimiter

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

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

3.11.6 Duplicating File Descriptors

The redirection operator

`[n]<&`*word*

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

The operator

`[n]>&`*word*

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

3.11.7 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.

3.12 Executing Commands

3.12.1 Command Search and 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.

1. 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 Section 3.8 [Shell Functions], page 12.
2. 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.
3. If the name is neither a shell function nor a builtin, and contains no slashes, Bash searches each element of `$PATH` for a directory containing an executable file by that name. Bash uses a hash table to remember the full filenames of executable files (see the description of `hash` in Section 4.1 [Bourne Shell Builtins], page 29) to avoid multiple `PATH` searches. A full search of the directories in `$PATH` is performed only if the command is not found in the hash table. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.
4. If the search is successful, or if the command name contains one or more slashes, the shell executes the named program. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments supplied, if any.
5. If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be *shell script* (see Section 3.13 [Shell Scripts], page 26).

3.12.2 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`.

Bash allows you to manipulate the environment in several ways. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for *export* to child processes. Executed commands inherit the environment. The `export` and '`declare`

-x' commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the `unset` command, plus any additions via the `export` and '`declare -x`' commands.

The environment for any simple command or function may be augmented temporarily by prefixing it with parameter assignments, as described in Section 3.9 [Shell Parameters], page 12. These assignment statements affect only the environment seen by that command.

If the '`-k`' flag is set (see Section 7.5 [The Set Builtin], page 60, then all parameter assignments are placed in the environment for a command, not just those that precede the command name.

When Bash invokes an external command, the variable '`$_`' is set to the full path name of the command and passed to that command in its environment.

3.12.3 Exit Status

For the purposes of the shell, a command which exits with a zero exit status has succeeded. A non-zero exit status indicates failure. This seemingly counter-intuitive scheme is used so there is one well-defined way to indicate success and a variety of ways to indicate various failure modes. When a command terminates on a fatal signal whose number is *n*, Bash uses the value 128+*n* as the exit status.

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

The exit status is used by the Bash conditional commands (see Section 3.6 [Conditional Constructs], page 10) and some of the list constructs (see Section 3.4 [Lists], page 8).

All of the Bash builtins return an exit status of zero if they succeed and a non-zero status on failure, so they may be used by the conditional and list constructs.

3.12.4 Signals

When Bash is interactive, it ignores `SIGTERM` (so that '`kill 0`' does not kill an interactive shell), and `SIGINT` is caught and handled (so that the `wait` builtin is interruptible). When Bash receives a `SIGINT`, it breaks out of any executing loops. In all cases, Bash ignores `SIGQUIT`. If job control is in effect (see Chapter 8 [Job Control], page 77), Bash ignores `SIGTTIN`, `SIGTTOU`, and `SIGTSTP`.

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

The shell exits by default upon receipt of a `SIGHUP`. Before exiting, it resends the `SIGHUP` to all jobs, running or stopped. To prevent the shell from sending the `SIGHUP` signal to a particular job, remove it from the jobs table with the `disown` builtin (see Section 8.2 [Job Control Builtins], page 78) or use `disown -h` to mark it to not receive `SIGHUP`.

3.13 Shell Scripts

A shell script is a text file containing shell commands. When such a file is used as the first non-option argument when invoking Bash, and neither the '`-c`' nor '`-s`' option is supplied (see Section 7.1 [Invoking Bash], page 49), Bash reads and executes commands from the file, then exits. This mode of operation creates a non-interactive shell. When Bash runs a shell script, it sets the special parameter `0` to the name of the file, rather than the name of the shell, and the positional parameters are set to the remaining arguments, if any are given. If no additional arguments are supplied, the positional parameters are unset.

A shell script may be made executable by using the `chmod` command to turn on the execute bit. When Bash finds such a file while searching the `$PATH` for a command, it spawns a subshell to execute it. In other words, executing

```
filename arguments
```

is equivalent to executing

```
bash filename arguments
```

if `filename` is an executable shell script. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to interpret the script.

Most versions of Unix make this a part of the kernel's command execution mechanism. If the first line of a script begins with the two characters '`#!`', the remainder of the line specifies an interpreter for the program. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the script file, followed by the name of the script file, followed by the rest of the arguments. Bash will perform this action on operating systems that do not handle it themselves. Note that some older versions of Unix limit the interpreter name and argument to a maximum of 32 characters.

4 Bourne Shell Style Features

This section briefly summarizes things which Bash inherits from the Bourne Shell: builtins, variables, and other features. It also lists the significant differences between Bash and the Bourne Shell.

4.1 Bourne Shell Builtins

The following shell builtin commands are inherited from the Bourne Shell. These commands are implemented as specified by the POSIX 1003.2 standard.

`:`

: `[`*arguments*`]`

Do nothing beyond expanding *arguments* and performing redirections.

`.`

. *filename*

Read and execute commands from the *filename* argument in the current shell context.

`break`

`break [`*n*`]`

Exit from a `for`, `while`, `until`, or `select` loop. If *n* is supplied, the *n*th enclosing loop is exited.

`cd`

`cd [-LP] [`*directory*`]`

Change the current working directory to *directory*. If *directory* is not given, the value of the `HOME` shell variable is used. If the shell variable `CDPATH` exists, it is used as a search path. If *directory* begins with a slash, `CDPATH` is not used. The '`-P`' option means to not follow symbolic links; symlinks are followed by default or with the '`-L`' option.

`continue`

`continue [`*n*`]`

Resume the next iteration of an enclosing `for`, `while`, `until`, or `select` loop. If *n* is supplied, the execution of the *n*th enclosing loop is resumed.

`eval`

`eval [`*arguments*`]`

The arguments are concatenated together into a single command, which is then read and executed.

`exec`

`exec [-cl] [-a `*name*`] [`*command*`] [`*arguments*`]`

If *command* is supplied, it replaces the shell. If the '`-l`' option is supplied, the shell places a dash in the zeroth arg passed to *command*. This is what the `login` program 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 *command*. If no *command* is specified, redirections may be used to affect the current shell environment.

`exit`

`exit [`*n*`]`

Exit the shell, returning a status of *n* to the shell's parent.

`export`

`export [-fn] [-p] [`*name*`[=`*value*`]]`

Mark each *name* to be passed to child processes in the environment. If the '`-f`' option is supplied, the *name*s refer to shell functions. The '`-n`' option means to no longer mark each *name* for export. If no *names* are supplied, or if the '`-p`' option is given, a list of exported names is displayed.

`getopts`

`getopts` *optstring name* `[`*args*`]`

`getopts` is used by shell scripts to parse positional parameters. *optstring* contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, `getopts` places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable `OPTIND`. `OPTIND` is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, `getopts` places that argument into the variable `OPTARG`. The shell does not reset `OPTIND` automatically; it must be manually reset between multiple calls to `getopts` within the same shell invocation if a new set of parameters is to be used.

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

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

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

`getopts` normally parses the positional parameters, but if more arguments are given in *args*, `getopts` parses those instead.

`hash`

`hash [-r] [-p` *filename*`] [`*name*`]`

Remember the full filenames of commands specified as arguments, so they need not be searched for on subsequent invocations. The commands are found by searching through the directories listed in `$PATH`. The '`-p`' option inhibits the path search, and *filename* is used as the location of *name*. The '`-r`' option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is printed.

`pwd`

`pwd [-LP]`

Print the current working directory. If the '`-P`' option is supplied, the path printed will not contain symbolic links. If the '`-L`' option is supplied, the path printed may contain symbolic links.

`readonly`

`readonly [-apf] [`*name*`]` ...

Mark each *name* as unchangable. The values of these names may not be changed by subsequent assignment. If the '`-f`' option is supplied, each *name* refers to a shell function. The '`-a`' option means each *name* refers to an array variable. If no *name* arguments are given, or if the '`-p`' option is supplied, a list of all readonly names is printed.

`return`

`return [`*n*`]`

Cause a shell function to exit with value *n*. This may also be used to terminate execution of a script being executed with the `.` builtin.

`shift`

`shift [`*n*`]`

Shift positional parameters to the left by *n*. The positional parameters from *n*`+`1 ... are renamed to `$1` ... . Parameters represented by the numbers `$#` to *n*`+`1 are unset. *n* must be a non-negative number less than or equal to `$#`.

`test`

`[` Evaluate a conditional expression (see Section 7.6 [Bash Conditional Expressions], page 62).

`times`

    times

Print out the user and system times used by the shell and its children.

`trap`

    trap [-lp] [arg] [sigspec ...]

The commands in *arg* are to be read and executed when the shell receives signal *sigspec*. If *arg* is absent or equal to '`-`', all specified signals are reset to the values they had when the shell was started. If *arg* is the null string, then the signal specified by each *sigspec* is ignored by the shell and commands it invokes. If *arg* is '`-p`', the shell displays the trap commands associated with each *sigspec*. If no arguments are supplied, or only '`-p`' is given, `trap` prints the list of commands associated with each signal number. Each *sigspec* is either a signal name such as `SIGINT` (with or without the `SIG` prefix) or a signal number. If a *sigspec* is `0` or `EXIT`, *arg* is executed when the shell exits. If a *sigspec* is `DEBUG`, the command *arg* is executed after every simple command. The '`-l`' option causes the shell to print a list of signal names and their corresponding numbers.

Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created.

`umask`

    umask [-S] [mode]

Set the shell process's file creation mask to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; if not, it is interpreted as a symbolic mode mask similar to that accepted by the `chmod` command. If *mode* is omitted, the current value of the mask is printed. If the '`-S`' option is supplied without a *mode* argument, the mask is printed in a symbolic format.

`unset`

    unset [-fv] [name]

Each variable or function *name* is removed. If no options are supplied, or the '`-v`' option is given, each *name* refers to a shell variable. If the '`-f`' option is given, the *name*s refer to shell functions, and the function definition is removed. Read-only variables and functions may not be unset.

4.2 Bourne Shell Variables

Bash uses certain shell variables in the same way as the Bourne shell. In some cases, Bash assigns a default value to the variable.

`IFS` A list of characters that separate fields; used when the shell splits words as part of expansion.

`PATH` A colon-separated list of directories in which the shell looks for commands.

`HOME` The current user's home directory; the default for the `cd` builtin command.

`CDPATH` A colon-separated list of directories used as a search path for the `cd` command.

`MAILPATH` A colon-separated list of files which the shell periodically checks for new mail. You can also specify what message is printed by separating the file name from the message with a '`?`'. When used in the text of the message, `$_` stands for the name of the current mailfile.

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

`PS1` The primary prompt string. The default value is '`\s-\v\$ `'.

`PS2` The secondary prompt string. The default value is '`> `'.

`OPTIND` The index of the last option processed by the `getopts` builtin.

`OPTARG` The value of the last option argument processed by the `getopts` builtin.

4.3 Other Bourne Shell Features

Bash implements essentially the same grammar, parameter and variable expansion, redirection, and quoting as the Bourne Shell. Bash uses the POSIX 1003.2 standard as the specification of how these features are to be implemented. There are some differences between the traditional Bourne shell and the POSIX standard; this section quickly details the differences of significance. A number of these differences are explained in greater depth in subsequent sections.

4.3.1 Major Differences From The SVR4.2 Bourne Shell

Bash is POSIX-conformant, even where the POSIX specification differs from traditional `sh` behavior.

Bash has multi-character invocation options (see Section 7.1 [Invoking Bash], page 49).

Bash has command-line editing (see Chapter 10 [Command Line Editing], page 85) and the `bind` builtin.

Bash has command history (see Section 9.1 [Bash History Facilities], page 81) and the `history` and `fc` builtins to manipulate it.

Bash implements `csh`-like history expansion (see Section 9.2 [History Interaction], page 82).

Bash has one-dimensional array variables (see Section 7.9 [Arrays], page 71), and the appropriate variable expansions and assignment syntax to use them. Some of the Bash builtins take options to act on arrays. Bash provides some built-in array variables.

Bash implements the `!` keyword to negate the return value of a pipeline (see Section 3.3 [Pipelines], page 8). Very useful when an `if` statement needs to act only if a test fails.

Bash includes the `select` compound command, which allows the generation of simple menus (see Section 6.1 [Korn Shell Constructs], page 45).

Bash includes brace expansion (see Section 5.1 [Brace Expansion], page 41) and tilde expansion (see Section 5.2 [Tilde Expansion], page 42).

Bash implements command aliases and the `alias` and `unalias` builtins (see Section 6.4 [Aliases], page 47).

Bash provides shell arithmetic and arithmetic expansion (see Section 7.8 [Shell Arithmetic], page 69).

The POSIX and `ksh`-style `$()` form of command substitution is implemented (see Section 3.10.2 [Command Substitution], page 17), and preferred to the Bourne shell's `` ` ` `` (which is also implemented for backwards compatibility).

Variables present in the shell's initial environment are automatically exported to child processes. The Bourne shell does not normally do this unless the variables are explicitly marked using the `export` command.

Bash includes the POSIX and `ksh`-style pattern removal '`%`', '`#`', '`%%`' and '`##`' constructs to remove leading or trailing substrings from variable values (see Section 3.10.1 [Shell Parameter Expansion], page 15).

The expansion `${#xx}`, which returns the length of `$xx`, is supported (see Section 3.10.1 [Shell Parameter Expansion], page 15).

The `$'...'` quoting syntax, which expands ANSI-C backslash-escaped characters in the text between the single quotes, is supported (see Section 3.1.2.4 [ANSI-C Quoting], page 7).

Bash supports the `$"..."` quoting syntax to do locale-specific translation of the characters between the double quotes. The '`-D`' and '`--dump-strings`' invocation options list the translatable strings found in a script (see Section 3.1.2.5 [Locale Translation], page 7).

The expansion `${var:`*offset*`[:`*length*`]}`, which expands to the substring of `var`'s value of length *length*, optionally beginning at *offset*, is present (see Section 3.10.1 [Shell Parameter Expansion], page 15).

The expansion `${var/[/]`*pattern*`[/`*replacement*`]}`, which matches *pattern* and replaces it with *replacement* in the value of `var`, is available (see Section 3.10.1 [Shell Parameter Expansion], page 15).

Bash has *indirect* variable expansion using `${!word}` (see Section 3.10.1 [Shell Parameter Expansion], page 15).

Bash can expand positional parameters beyond `$9` using `${`*num*`}`.

Bash has process substitution (see Section 3.10.3 [Process Substitution], page 18).

Bash automatically assigns variables that provide information about the current user (`UID`, `EUID`, and `GROUPS`), the current host (`HOSTTYPE`, `OSTYPE`, `MACHTYPE`, and `HOSTNAME`), and the instance of Bash that is running (`BASH`, `BASH_VERSION`, and `BASH_VERSINFO`. See Section 7.7 [Bash Variables], page 65, for details.

The `IFS` variable is used to split only the results of expansion, not all words (see Section 3.10.4 [Word Splitting], page 19). This closes a longstanding shell security hole.

It is possible to have a variable and a function with the same name; `sh` does not separate the two name spaces.

Bash functions are permitted to have local variables using the `local` builtin, and thus useful recursive functions may be written.

Variable assignments preceding commands affect only that command, even builtins and functions (see Section 3.12.2 [Environment], page 24). In `sh`, all variable assignments preceding commands are global unless the command is executed from the file system.

Bash performs filename expansion on filenames specified as operands to output redirection operators.

Bash contains the '`<>`' redirection operator, allowing a file to be opened for both reading and writing, and the '`&>`' redirection operator, for directing standard output and standard error to the same file (see Section 3.11 [Redirections], page 20).

The `noclobber` option is available to avoid overwriting existing files with output redirection (see Section 7.5 [The Set Builtin], page 60). The '`>|`' redirection operator may be used to override `noclobber`.

Bash interprets special backslash-escaped characters in the prompt strings when interactive (see Section 7.10 [Printing a Prompt], page 72).

Bash allows you to write a function to override a builtin, and provides access to that builtin's functionality within the function via the `builtin` and `command` builtins (see Section 7.4 [Bash Builtins], page 53).

The `command` builtin allows selective disabling of functions when command lookup is performed (see Section 7.4 [Bash Builtins], page 53).

Individual builtins may be enabled or disabled using the `enable` builtin (see Section 7.4 [Bash Builtins], page 53).

The Bash `hash` builtin allows a name to be associated with an arbitrary filename, even when that filename cannot be found by searching the `$PATH`, using '`hash -p`'.

Shell functions may be exported to children via the environment (see Section 3.8 [Shell Functions], page 12).

Bash includes a `help` builtin for quick reference to shell facilities (see Section 7.4 [Bash Builtins], page 53).

The Bash `read` builtin (see Section 7.4 [Bash Builtins], page 53) will read a line ending in '\' with the '-r' option, and will use the `REPLY` variable as a default if no arguments are supplied. The Bash `read` builtin also accepts a prompt string with the '-p' option and will use Readline to obtain the line when given the '-e' option.

Bash includes the `shopt` builtin, for finer control of shell optional capabilities (see Section 7.4 [Bash Builtins], page 53).

Bash has much more optional behavior controllable with the `set` builtin (see Section 7.5 [The Set Builtin], page 60).

The `disown` builtin can remove a job from the internal shell job table (see Section 8.2 [Job Control Builtins], page 78).

The `return` builtin may be used to abort execution of scripts executed with the `.` or `source` builtins (see Section 4.1 [Bourne Shell Builtins], page 29).

The `test` builtin (see Section 4.1 [Bourne Shell Builtins], page 29) is slightly different, as it implements the POSIX 1003.2 algorithm, which specifies the behavior based on the number of arguments.

The `trap` builtin (see Section 4.1 [Bourne Shell Builtins], page 29) allows a `DEBUG` pseudo-signal specification, similar to `EXIT`. Commands specified with a `DEBUG` trap are executed after every simple command. The `DEBUG` trap is not inherited by shell functions.

The Bash `export`, `readonly`, and `declare` builtins can take a '-f' option to act on shell functions, a '-p' option to display variables with various attributes set in a format that can be used as shell input, a '-n' option to remove various variable attributes, and 'name=value' arguments to set variable attributes and values simultaneously.

The Bash `cd` and `pwd` builtins (see Section 4.1 [Bourne Shell Builtins], page 29) each take '-L' and '-P' builtins to switch between logical and physical modes.

The Bash `type` builtin is more extensive and gives more information about the names it finds (see Section 7.4 [Bash Builtins], page 53).

Bash implements a `csh`-like directory stack, and provides the `pushd`, `popd`, and `dirs` builtins to manipulate it (see Section 5.3 [C Shell Builtins], page 42). Bash also makes the directory stack visible as the value of the `DIRSTACK` shell variable.

The Bash restricted mode is more useful (see Section 7.11 [The Restricted Shell], page 73); the SVR4.2 shell restricted mode is too limited.

Bash has the `time` reserved word and command timing (see Section 3.3 [Pipelines], page 8). The display of the timing statistics may be controlled with the `TIMEFORMAT` variable.

The SVR4.2 shell has two privilege-related builtins (`mldmode` and `priv`) not present in Bash.

Bash does not have the `stop` or `newgrp` builtins.

Bash does not use the `SHACCT` variable or perform shell accounting.

The SVR4.2 `sh` uses a `TIMEOUT` variable like Bash uses `TMOUT`.

More features unique to Bash may be found in Chapter 7 [Bash Features], page 49.

4.3.2 Implementation Differences From The SVR4.2 Shell

Since Bash is a completely new implementation, it does not suffer from many of the limitations of the SVR4.2 shell. For instance:

- Bash does not fork a subshell when redirecting into or out of a shell control structure such as an `if` or `while` statement.
- Bash does not allow unbalanced quotes. The SVR4.2 shell will silently insert a needed closing quote at `EOF` under certain circumstances. This can be the cause of some hard-to-find errors.
- The SVR4.2 shell uses a baroque memory management scheme based on trapping `SIGSEGV`. If the shell is started from a process with `SIGSEGV` blocked (e.g., by using the `system()` C library function call), the shell misbehaves badly.
- In a questionable attempt at security, the SVR4.2 shell, when invoked without the '`-p`' option, will alter its real and effective UID and GID if they are less than some magic threshold value, commonly 100. This can lead to unexpected results.
- The SVR4.2 shell does not allow users to trap `SIGALRM` or `SIGCHLD`.
- For some reason, the SVR4.2 shell does not allow the `MAILCHECK` variable to be unset.

- The SVR4.2 shell treats '^' as the undocumented equivalent of '|'.
- Bash allows multiple option arguments when it is invoked (`-x -v`); the SVR4.2 shell allows only one option argument (`-xv`). In fact, some versions of the shell dump core if the second argument begins with a '-'.
- The SVR4.2 shell exits a script if any builtin fails; Bash exits a script only if one of the POSIX.2 special builtins fails, and only for certain failures, as enumerated in the POSIX.2 standard.
- The SVR4.2 shell behaves differently when invoked as `jsh` (it turns on job control).

5 C-Shell Style Features

The C-Shell (`csh`) was created by Bill Joy at The University of California at Berkeley. It is generally considered to have better features for interactive use than the original Bourne shell. Some of the `csh` features present in Bash include job control, history expansion, 'protected' redirection, and several variables to control the interactive behaviour of the shell (e.g., `IGNOREEOF`).

See Chapter 9 [Using History Interactively], page 81, for details on history expansion.

5.1 Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *filename expansion* (see Section 3.10.5 [Filename Expansion], page 19), but the file names generated need not exist. Patterns to be brace expanded take the form of an optional *preamble*, followed by a series of comma-separated strings between a pair of braces, followed by an optional *postamble*. The preamble is prepended to each string contained within the braces, and the postamble is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example,

```
bash$ echo a{d,c,b}e
ade ace abe
```

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

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

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

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

or

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

5.2 Tilde Expansion

Bash has tilde (~) expansion, similar, but not identical, to that of `csh`. The following table shows what unquoted words beginning with a tilde expand to.

`~` The current value of `$HOME`.

`~/foo` '`$HOME/foo`'

`~fred/foo`
: The subdirectory `foo` of the home directory of the user `fred`.

`~+/foo` '`$PWD/foo`'

`~-/foo` '`$OLDPWD/foo`'

Bash will also tilde expand words following redirection operators and words following '`=`' in assignment statements.

5.3 C Shell Builtins

Bash has several builtin commands whose definition is very similar to `csh`.

`pushd`

`pushd [`*dir*` | +`*N*` | -`*N*`] [-n]`

Save the current directory on a list and then `cd` to *dir*. With no arguments, exchanges the top two directories.

`+`*N* Brings the *N*th directory (counting from the left of the list printed by `dirs`, starting with zero) to the top of the list by rotating the stack.

`-`*N* Brings the *N*th directory (counting from the right of the list printed by `dirs`, starting with zero) to the top of the list by rotating the stack.

`-n` Suppresses the normal change of directory when adding directories to the stack, so that only the stack is manipulated.

dir Makes the current working directory be the top of the stack, and then `cd`s to *dir*. You can see the saved directory list with the `dirs` command.

`popd`

    popd [+N | -N] [-n]

Pop the directory stack, and `cd` to the new top directory. When no arguments are given, `popd` removes the top directory from the stack and performs a `cd` to the new top directory. The elements are numbered from 0 starting at the first directory listed with `dirs`; i.e., `popd` is equivalent to `popd +0`.

`+`*N* Removes the *N*th directory (counting from the left of the list printed by `dirs`), starting with zero.

`-`*N* Removes the *N*th directory (counting from the right of the list printed by `dirs`), starting with zero.

`-n` Suppresses the normal change of directory when removing directories from the stack, so that only the stack is manipulated.

`dirs`

    dirs [+N | -N] [-clvp]

Display the list of currently remembered directories. Directories find their way onto the list with the `pushd` command; you can get back up through the list with the `popd` command.

`+`*N* Displays the *N*th directory (counting from the left of the list printed by `dirs` when invoked without options), starting with zero.

`-`*N* Displays the *N*th directory (counting from the right of the list printed by `dirs` when invoked without options), starting with zero.

`-c` Clears the directory stack by deleting all of the elements.

`-l` Produces a longer listing; the default listing format uses a tilde to denote the home directory.

`-p` Causes `dirs` to print the directory stack with one entry per line.

`-v` Causes `dirs` to print the directory stack with one entry per line, prepending each entry with its index in the stack.

`history`

    history [-c] [n]
    history [-anrw] [filename]
    history -ps arg

Display the history list with line numbers. Lines prefixed with with a '`*`' have been modified. An argument of *n* says to list only the last *n* lines. Options, if supplied, have the following meanings:

`-w` Write out the current history to the history file.

`-r` Read the current history file and append its contents to the history list.

`-a` Append the new history lines (history lines entered since the beginning of the current Bash session) to the history file.

`-n` Append the history lines not already read from the history file to the current history list. These are lines appended to the history file since the beginning of the current Bash session.

`-c` Clear the history list. This may be combined with the other options to replace the history list completely.

`-s` The *args* are added to the end of the history list as a single entry.

`-p` Perform history substitution on the *args* and display the result on the standard output, without storing the results in the history list.

When the '`-w`', '`-r`', '`-a`', or '`-n`' option is used, if *filename* is given, then it is used as the history file. If not, then the value of the `HISTFILE` variable is used.

`logout` Exit a login shell.

`source` A synonym for `.` (see Section 4.1 [Bourne Shell Builtins], page 29).

5.4 C Shell Variables

`IGNOREEOF`

If this variable is set, its value is used the number of consecutive `EOF`s Bash will read before exiting. By default, Bash will exit upon reading a single `EOF`. If `IGNOREEOF` is not set to a numeric value, Bash acts as if its value were 10.

6 Korn Shell Style Features

This section describes features primarily inspired by the Korn Shell (`ksh`). In some cases, the POSIX 1003.2 standard has adopted these commands and variables from the Korn Shell; Bash implements those features using the POSIX standard as a guide.

6.1 Korn Shell Constructs

Bash includes the Korn Shell `select` construct. This construct allows the easy generation of menus. It has almost the same syntax as the `for` command.

The syntax of the `select` command is:

```
select name [in words ...]; do commands; done
```

The list of words following `in` is expanded, generating a list of items. The set of expanded words is printed on the standard error, each preceded by a number. If the '`in` *words*' is omitted, the positional parameters are printed. The `PS3` prompt is then displayed and a line is 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 `select` command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable `REPLY`.

The *commands* are executed after each selection until a `break` or `return` command is executed, at which point the `select` command completes.

Bash also has adopted command timing from the Korn shell. If the `time` reserved word precedes a pipeline, which may consist of a single command, timing statistics for the pipeline are displayed when it completes. The statistics currently consist of elapsed (wall-clock) time and user and system time consumed by the command's execution.

The use of `time` as a reserved word permits the timing of shell builtins, shell functions, and pipelines. An external `time` command cannot time these easily.

6.2 Korn Shell Builtins

This section describes Bash builtin commands taken from `ksh`.

`fc`

```
fc [-e ename] [-nlr] [first] [last]
fc -s [pat=rep] [command]
```

Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. Both *first* and *last* may be specified as a string (to locate the most recent command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified it is set to *first*. If *first* is not specified it is set to the previous command for editing and −16 for listing. If the '`-l`' flag is given, the commands are listed on standard output. The '`-n`' flag suppresses the command numbers when listing. The '`-r`' flag reverses the order of the listing. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the following variable expansion is used: `${FCEDIT:-${EDITOR:-vi}}`. This says to use the value of the `FCEDIT` variable if set, or the value of the `EDITOR` variable if that is set, or `vi` if neither is set. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after each instance of *pat* in the selected command is replaced by *rep*.

A useful alias to use with the `fc` command is `r='fc -s'`, so that typing '`r cc`' runs the last command beginning with `cc` and typing '`r`' re-executes the last command (see Section 6.4 [Aliases], page 47).

`let` The `let` builtin allows arithmetic to be performed on shell variables. For details, refer to Section 7.8.3 [Arithmetic Builtins], page 71.

`typeset` The `typeset` command is supplied for compatibility with the Korn shell; however, it has been deprecated in favor of the `declare` command (see Section 7.4 [Bash Builtins], page 53).

6.3 Korn Shell Variables

`REPLY` The default variable for the `read` builtin.

`RANDOM` Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to this variable seeds the random number generator.

SECONDS This variable expands to the number of seconds since the shell was started. Assignment to this variable resets the count to the value assigned, and the expanded value becomes the value assigned plus the number of seconds since the assignment.

PS3 The value of this variable is used as the prompt for the `select` command. If this variable is not set, the `select` command prompts with '`#? `'

PS4 This is the prompt printed before the command line is echoed when the '`-x`' option is set (see Section 7.5 [The Set Builtin], page 60). The default is '`+ `'.

PWD The current working directory as set by the `cd` builtin.

OLDPWD The previous working directory as set by the `cd` builtin.

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

LINENO The line number in the script or shell function currently executing.

FCEDIT The editor used as a default by the `fc` builtin command.

6.4 Aliases

The shell maintains a list of *aliases* that may be set and unset with the `alias` and `unalias` builtin commands.

The first word of each command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The alias name and the replacement text may contain any valid shell input, including shell metacharacters, with the exception that the alias name may not contain `=`. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias `ls` to `"ls -F"`, for instance, and Bash does not try to recursively expand the replacement text. If the last character of the alias value is a space or tab character, then the next command word following the alias is also checked for alias expansion.

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

There is no mechanism for using arguments in the replacement text, as in `csh`. If arguments are needed, a shell function should be used (see Section 3.8 [Shell Functions], page 12).

Aliases are not expanded when the shell is not interactive, unless the `expand_aliases` shell option is set using `shopt` (see Section 7.4 [Bash Builtins], page 53).

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

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

6.4.1 Alias Builtins

`alias`

> `alias [-p] [`*name*`[=`*value*`] …]`
>
> Without arguments or with the '`-p`' option, `alias` prints the list of aliases on the standard output in a form that allows them to be reused as input. If arguments are supplied, an alias is defined for each *name* whose *value* is given. If no *value* is given, the name and value of the alias is printed.

`unalias`

> `unalias [-a] [`*name* `… ]`
>
> Remove each *name* from the list of aliases. If '`-a`' is supplied, all aliases are removed.

7 Bash Features

This section describes features unique to Bash.

7.1 Invoking Bash

```
bash [long-opt] [-ir] [-abefhkmnptuvxdBCDHP] [-o option] [argument ...]
bash [long-opt] [-abefhkmnptuvxdBCDHP] [-o option] -c string [argument ...]
bash [long-opt] -s [-abefhkmnptuvxdBCDHP] [-o option] [argument ...]
```

In addition to the single-character shell command-line options (see Section 7.5 [The Set Builtin], page 60), there are several multi-character options that you can use. These options must appear on the command line before the single-character options in order for them to be recognized.

`--dump-strings`
: Equivalent to '`-D`'.

`--help`
: Display a usage message on standard output and exit sucessfully.

`--login`
: Make this shell act as if it were directly invoked by login. This is equivalent to '`exec -l bash`' but can be issued from another shell, such as `csh`. If you wanted to replace your current login shell with a Bash login shell, you would say '`exec bash --login`'.

`--noediting`
: Do not use the GNU Readline library (see Chapter 10 [Command Line Editing], page 85) to read interactive command lines.

`--noprofile`
: Don't load the system-wide startup file '`/etc/profile`' or any of the personal initialization files '`~/.bash_profile`', '`~/.bash_login`', or '`~/.profile`' when Bash is invoked as a login shell.

`--norc`
: Don't read the '`~/.bashrc`' initialization file in an interactive shell. This is on by default if the shell is invoked as `sh`.

`--posix`
: Change the behavior of Bash where the default operation differs from the POSIX 1003.2 standard to match the standard. This is intended to make Bash behave as a strict superset of that standard. See Section 7.12 [Bash POSIX Mode], page 74, for a description of the Bash POSIX mode.

`--rcfile` *filename*
: Execute commands from *filename* (instead of '`~/.bashrc`') in an interactive shell.

`--restricted`
: Make the shell a restricted shell (see Section 7.11 [The Restricted Shell], page 73).

`--verbose`
: Equivalent to '`-v`'.

`--version`
: Show version information for this instance of Bash on the standard output and exit successfully.

There are several single-character options you can give which are not available with the `set` builtin.

`-c` *string*
: Read and execute commands from *string* after processing the options, then exit. Any remaining arguments are assigned to the positional parameters, starting with `$0`.

`-i`
: Force the shell to run interactively.

`-r`
: Make the shell restricted.

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

`-D`
: A list of all double-quoted strings preceded by '`$`' is printed on the standard ouput. These are the strings that are subject to language translation when the current locale is not `C` or `POSIX` (see Section 3.1.2.5 [Locale Translation], page 7). This implies the '`-n`' option; no commands will be executed.

An *interactive* shell is one whose input and output are both connected to terminals (as determined by `isatty()`), or one started with the '`-i`' option.

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 (see Section 3.13 [Shell Scripts], page 26). When Bash is invoked in this fashion, `$0` is set to the name of the file, and the positional parameters are set to the remaining arguments. Bash reads and executes commands from this file, then exits. Bash's exit status is the exit status of the last command executed in the script. If no commands are executed, the exit status is 0.

7.2 Bash Startup Files

This section describs how Bash executes its startup files. If any of the files exist but cannot be read, Bash reports an error. Tildes are expanded in file names as described above under Tilde Expansion (see Section 5.2 [Tilde Expansion], page 42).

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

When a login shell exits, Bash reads and executes commands from the file '~/.bash_logout', if it exists.

When an interactive shell that is not a login shell is started, Bash reads and executes commands from '~/.bashrc', if that file exists. 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'.

So, typically, your '~/.bash_profile' contains the line

```
if [ -f '~/.bashrc' ]; then . '~/.bashrc'; fi
```

after (or before) any login-specific initializations.

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

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

but 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 behavior of historical versions of sh as closely as possible, while conforming to the POSIX standard as well.

When invoked as a login shell, it first attempts to read and execute commands from '/etc/profile' and '~/.profile', in that order. The '--noprofile' option may be used to inhibit this behavior. When invoked as an interactive shell with the name sh, bash looks for the variable ENV, expands its value if it is defined, and uses the expanded value as the name of a file to read and execute. Since a shell invoked as sh does not attempt to read and execute commands from any other startup files, the '--rcfile' option has no effect. A non-interactive shell invoked with the name sh does not attempt to read any 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, the ENV variable is expanded and commands are read and executed from the file whose name is the expanded value. No other startup files are read. This is done by interactive shells only.

Bash attempts to determine when it is being run by the remote shell daemon, usually rshd. If Bash determines it is being run by rshd, 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 rshd does not generally invoke the shell with those options or allow them to be specified.

7.3 Is This Shell Interactive?

As defined in Section 7.1 [Invoking Bash], page 49, an interactive shell is one whose input and output are both connected to terminals (as determined by isatty(3)), or one started with the '-i' option.

You may wish to determine within a startup script whether Bash is running interactively or not. To do this, examine the variable $PS1; it is unset in non-interactive shells, and set in interactive shells. Thus:

```
if [ -z "$PS1" ]; then
        echo This shell is not interactive
else
        echo This shell is interactive
fi
```

Alternatively, you may test the value of the '-' special parameter. It contains i when the shell is interactive. For example:

```
case "$-" in
*i*) echo This shell is interactive ;;
*) echo This shell is not interactive ;;
esac
```

7.4 Bash Builtin Commands

This section describes builtin commands which are unique to or have been extended in Bash.

`bind`

`bind [-m` *keymap*`] [-lpsvPSV] [-q` *name*`] [-r` *keyseq*`]`
`bind [-m` *keymap*`] -f` *filename*
`bind [-m` *keymap*`]` *keyseq:function-name*

Display current Readline (see Chapter 10 [Command Line Editing], page 85) key and function bindings, or bind a key sequence to a Readline function or macro. The binding syntax accepted is identical to that of '`.inputrc`' (see Section 10.3 [Readline Init File], page 89), but each binding must be passed as a separate argument: e.g., '`"\C-x\C-r":re-read-init-file`'. Options, if supplied, have the following meanings:

`-m` *keymap*
: Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`.

`-l`
: List the names of all Readline functions

`-p`
: Display Readline function names and bindings in such a way that they can be re-read

`-P`
: List current Readline function names and bindings

`-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

`-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

`-f` *filename*
: Read key bindings from *filename*

`-q` Query about which keys invoke the named *function*

`-r` *keyseq* Remove any current binding for *keyseq*

`builtin`

`builtin [`*shell-builtin* `[`*args*`]]`

Run a shell builtin. This is useful when you wish to define a shell function with the same name as a shell builtin, but need the functionality of the builtin within the function itself.

`command`

`command [-pVv]` *command* `[`*args* ...`]`

Runs *command* with *arg* ignoring shell functions. If you have a shell function called `ls`, and you wish to call the command `ls`, you can say '`command ls`'. The '`-p`' option means to use a default value for `$PATH` that is guaranteed to find all of the standard utilities.

If either the '`-V`' or '`-v`' option is supplied, a description of *command* is printed. The '`-v`' option causes a single word indicating the command or file name used to invoke *command* to be printed; the '`-V`' option produces a more verbose description.

`declare`

`declare [-afFrxi] [-p] [`*name*`[=`*value*`]]`

Declare variables and give them attributes. If no *name*s are given, then display the values of variables instead.

The '`-p`' option will display the attributes and values of each *name*. When '`-p`' is used, additional options are ignored. The '`-F`' option inhibits the display of function definitions; only the function name and attributes are printed. '`-F`' implies '`-f`'. The following options can be used to restrict output to variables with the specified attributes or to give variables attributes:

`-a` Each *name* is an array variable (see Section 7.9 [Arrays], page 71).

`-f` Use function names only.

`-i` The variable is to be treated as an integer; arithmetic evaluation (see Section 7.8 [Shell Arithmetic], page 69) is performed when the variable is assigned a value.

`-r` Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements.

`-x` Mark each *name* for export to subsequent commands via the environment.

Using '`+`' instead of '`-`' turns off the attribute instead. When used in a function, `declare` makes each *name* local, as with the `local` command.

`echo`

```
echo [-neE] [arg ...]
```

Output the `args`, separated by spaces, terminated with a newline. The return status is always 0. If '`-n`' is specified, the trailing newline is suppressed. If the '`-e`' option is given, interpretation of the following backslash-escaped characters is enabled. The '`-E`' option disables the interpretation of these escape characters, even on systems where they are interpreted by default. `echo` interprets the following escape sequences:

`\a` alert (bell)

`\b` backspace

`\c` suppress trailing newline

`\e` escape

`\f` form feed

`\n` new line

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\\` backslash

`\nnn` the character whose ASCII code is `nnn` (octal)

`enable`

`enable [-n] [-p] [-f` *filename*`] [-ads] [`*name* ...`]`

Enable and disable builtin shell commands. This allows you to use a disk command which has the same name as a shell builtin. If '`-n`' is used, the *name*s become disabled. Otherwise *name*s are enabled. For example, to use the `test` binary found via `$PATH` instead of the shell builtin version, type '`enable -n test`'.

If the '`-p`' option is supplied, or no *name* arguments appear, a list of shell builtins is printed. With no other arguments, the list consists of all enabled shell builtins. The '`-a`' option means to list each builtin with an indication of whether or not it is enabled.

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 loaded with '`-f`'. If there are no options, a list of the shell builtins is displayed. The '`-s`' option restricts `enable` to the POSIX.2 special builtins. If '`-s`' is used with '`-f`', the new builtin becomes a special builtin.

`help`

`help [`*pattern*`]`

Display helpful information about builtin commands. If *pattern* is specified, `help` gives detailed help on all commands matching *pattern*, otherwise a list of the builtins is printed.

`local`

`local` *name*`[=`*value*`]`

For each argument, create a local variable called *name*, and give it *value*. `local` can only be used within a function; it makes the variable *name* have a visible scope restricted to that function and its children.

`logout`

`logout [`*n*`]`

Exit a login shell, returning a status of *n* to the shell's parent.

`read`

`read [-a` *aname*`] [-p` *prompt*`] [-er] [`*name* ...`]`

One line is read from the standard input, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words assigned to the last *name*. Only the characters in the value of the `IFS` variable are recognized as word delimiters. If no names are supplied, the line read is assigned to the variable `REPLY`. The return code is zero, unless end-of-file is encountered. Options, if supplied, have the following meanings:

`-r` If this option is given, a backslash-newline pair is not ignored, and the backslash is considered to be part of the line.

`-p` *prompt*

Display `prompt`, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.

`-a` *aname* The words are assigned to sequential indices of the array variable *aname*, starting at 0.

`-e` Readline (see Chapter 10 [Command Line Editing], page 85) is used to obtain the line.

`shopt`

`shopt [-pqsu] [-o] [`*optname* ...`]`

Toggle the values of variables controlling optional shell behavior. 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. Other options have the following meanings:

`-s` Enable (set) each *optname*

`-u` Disable (unset) each *optname*.

`-q` Suppresses normal output; the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are given with '`-q`', the return status is zero if all *optnames* are enabled; non-zero otherwise.

`-o` Restricts the values of *optname* to be those defined for the '`-o`' option to the `set` builtin (see Section 7.5 [The Set Builtin], page 60).

If either of '`-s`' or '`-u`' is used with no *optname* arguments, the display is limited to those options which are set or unset, respectively.

Unless otherwise noted, the `shopt` options are disabled (off) by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a legal shell option.

The list of `shopt` options is:

`cdable_vars`
: If this is set, an argument to the `cd` builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.

`cdspell` If set, minor errors in the spelling of a directory component in a `cd` command will be corrected. The errors checked for are transposed characters, a missing character, and a character too many. If a correction is found, the corrected path is printed, and the command proceeds. This option is only used by interactive shells.

`checkhash`
: If this is set, Bash checks that a command found in the hash table exists before trying to execute it. If a hashed command no longer exists, a normal path search is performed.

`checkwinsize`
: If set, Bash checks the window size after each command and, if necessary, updates the values of `LINES` and `COLUMNS`.

`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.

`dotglob` If set, Bash includes filenames beginning with a '.' in the results of filename expansion.

`execfail` If this is set, a non-interactive shell will not exit if it cannot execute the file specified as an argument to the `exec` builtin command. An interactive shell does not exit if `exec` fails.

`histappend`
: If set, the history list is appended to the file named by the value of the `HISTFILE` variable when the shell exits, rather than overwriting the file.

`histreedit`
: If set, and Readline is being used, a user is given the opportunity to re-edit a failed history substitution.

`histverify`
: If set, and Readline is being used, the results of history substitution are not immediately passed to the shell parser. Instead, the resulting line is loaded into the Readline editing buffer, allowing further modification.

`hostcomplete`
: If set, and Readline is being used, Bash will attempt to perform hostname completion when a word beginning with '`@`' is being completed (see Section 10.4.6 [Commands For Completion], page 100). This option is enabled by default.

`interactive_comments`
: Allow a word beginning with '`#`' to cause that word and all remaining characters on that line to be ignored in an interactive shell. This option is enabled by default.

`lithist`
: If enabled, and the `cmdhist` option is enabled, multi-line commands are saved to the history with embedded newlines rather than using semicolon separators where possible.

`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 displayed.

`nullglob`
: If set, Bash allows filename patterns which match no files to expand to a null string, rather than themselves.

`promptvars`
: If set, prompt strings undergo variable and parameter expansion after being expanded (see Section 7.10 [Printing a Prompt], page 72). This option is enabled by default.

`shift_verbose`
: If this is set, the `shift` builtin prints an error message when the shift count exceeds the number of positional parameters.

`sourcepath`
: If set, the `source` builtin uses the value of `PATH` to find the directory containing the file supplied as an argument. This is enabled by default.

`type`

`type [-all] [-type | -path] [`*name* ...`]`

For each *name*, indicate how it would be interpreted if used as a command name.

If the '-type' flag is used, `type` returns a single word which is one of 'alias', 'function', 'builtin', 'file' or 'keyword', if *name* is an alias, shell function, shell builtin, disk file, or shell reserved word, respectively. If the *name* is not found, then nothing is printed, and `type` returns a failure status.

If the '-path' flag is used, `type` either returns the name of the disk file that would be executed, or nothing if '-type' would not return 'file'.

If the '-all' flag is used, returns all of the places that contain an executable named *file*. This includes aliases and functions, if and only if the '-path' flag is not also used.

`type` accepts '-a', '-t', and '-p' as equivalent to '-all', '-type', and '-path', respectively.

`ulimit`

```
ulimit [-acdflmnpstuvSH] [limit]
```

`ulimit` provides control over the resources available to processes started by the shell, on systems that allow such control. If an option is given, it is interpreted as follows:

`-S` change and report the soft limit associated with a resource.

`-H` change and report the hard limit associated with a resource.

`-a` all current limits are reported.

`-c` the maximum size of core files created.

`-d` the maximum size of a process's data segment.

`-f` the maximum size of files created by the shell.

`-l` The maximum size that may be locked into memory.

`-m` the maximum resident set size.

`-n` the maximum number of open file descriptors.

`-p` the pipe buffer size.

`-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 process.

If *limit* is given, it is the new value of the specified resource. Otherwise, the current value of the soft limit for the specified resource is printed, unless the '-H' option is supplied. When setting new limits, if neither '-H' nor '-S' is supplied, both the hard and soft limits are set. If no option is given, then '-f' is assumed. Values are in 1024-byte increments, except for '-t', which is in seconds, '-p', which is in units of 512-byte blocks, and '-n' and '-u', which are unscaled values.

7.5 The Set Builtin

This builtin is so overloaded that it deserves its own section.

`set`

    set [-abefhkmnptuvxdBCHP] [-o option] [argument ...]

`-a` Mark variables which are modified or created for export.

`-b` Cause the status of terminated background jobs to be reported immediately, rather than before printing the next primary prompt.

`-e` Exit immediately if a simple command exits with a non-zero status.

`-f` Disable file name generation (globbing).

`-h` Locate and remember (hash) commands as they are looked up for execution.

`-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` Job control is enabled (see Chapter 8 [Job Control], page 77).

`-n` Read commands but do not execute them.

`-o` *option-name*
Set the flag corresponding to *option-name*:

`allexport`
same as `-a`.

`braceexpand`
same as `-B`.

`emacs` use an `emacs`-style line editing interface (see Chapter 10 [Command Line Editing], page 85).

`errexit` same as `-e`.

`hashall` same as `-h`.

`histexpand`
same as `-H`.

`history` Enable command history, as described in Section 9.1 [Bash History Facilities], page 81. This option is on by default in interactive shells.

`ignoreeof`
the shell will not exit upon reading EOF.

`keyword` same as `-k`.

`monitor` same as `-m`.

`noclobber`
same as `-C`.

`noexec` same as `-n`.

`noglob` same as `-f`.

`notify` same as `-b`.

`nounset` same as `-u`.

`onecmd` same as `-t`.

`physical` same as `-P`.

`posix` change the behavior of Bash where the default operation differs from the POSIX 1003.2 standard to match the standard. This is intended to make Bash behave as a strict superset of that standard.

`privileged`
same as `-p`.

`verbose` same as `-v`.

`vi` use a `vi`-style line editing interface.

`xtrace` same as `-x`.

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

`-t` Exit after reading and executing one command.

`-u` Treat unset variables as an error when substituting.

`-v` Print shell input lines as they are read.

`-x` Print commands and their arguments as they are executed.

`-B` The shell will perform brace expansion (see Section 5.1 [Brace Expansion], page 41). This option is on by default.

`-C` Disallow output redirection to existing files.

`-H` Enable '`!`' style history substitution (see Section 9.2 [History Interaction], page 82). This flag is on by default for interactive shells.

`-P` If set, do not follow symbolic links when performing commands such as `cd` which change the current directory. The physical directory is used instead. By default, Bash follows the logical chain of directories when performing commands which change the current directory.

For example, if '`/usr/sys`' is a link to '`/usr/local/sys`' then:

```
$ cd /usr/sys; echo $PWD
/usr/sys
$ cd ..; pwd
/usr
```

If `set -P` is on, then:

```
$ cd /usr/sys; echo $PWD
/usr/local/sys
$ cd ..; pwd
/usr/local
```

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

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

Using '`+`' rather than '`-`' causes these flags to be turned off. The flags can also be used upon invocation of the shell. The current set of flags may be found in `$-`.

The remaining N *arguments* are positional parameters and are assigned, in order, to `$1`, `$2`, ... `$N`. If no arguments are given, all shell variables are printed.

7.6 Bash Conditional Expressions

Conditional expressions are used by the `test` and `[` builtins.

Expressions may be unary or binary. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Each operator and operand must be a separate argument. If *file* is of the form '`/dev/fd/`*N*', then file descriptor *N* is checked. Expressions are composed of the following primaries:

`-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.

`-k` *file* True if *file* has its "sticky" bit set.

`-L` *file* True if *file* exists and is a symbolic link.

`-p` *file* True if *file* exists and is a named pipe.

`-r` *file* True if *file* exists and is readable.

`-s` *file* True if *file* exists and has a size greater than zero.

`-S` *file* True if *file* exists and is a socket.

`-t` *fd* True if *fd* is opened on a terminal.

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

`-w` *file* True if *file* exists and is writable.

`-x` *file* True if *file* exists and is executable.

`-O` *file* True if *file* exists and is owned by the effective user id.

`-G` *file* True if *file* exists and is owned by the effective group id.

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

file1 `-ot` *file2*
True if *file1* is older than *file2*.

file1 `-ef` *file2*
True if *file1* and *file2* have the same device and inode numbers.

`-o` *optname*
True if shell option *optname* is enabled. The list of options appears in the description of the '`-o`' option to the `set` builtin (see Section 7.5 [The Set Builtin], page 60).

`-z` *string* True if the length of *string* is zero.

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

string1 `=` *string2*
True if the strings are equal. '`==`' may be used in place of '`=`'.

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.

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

expr1 `-a` *expr2*
: True if both *expr1* and *expr2* are true.

expr1 `-o` *expr2*
: True if either *expr1* and *expr2* is true.

arg1 `OP` *arg2*
: `OP` is one of '`-eq`', '`-ne`', '`-lt`', '`-le`', '`-gt`', or '`-ge`'. These arithmetic binary operators return true if *arg1* is equal 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.

The Bash `test` and `[` builtins evaluate conditional expressions using a set of rules based on the number of arguments. These are the rules:

0 arguments
: The expression is false.

1 argument
: The expression is true if and only if the argument is not null.

2 arguments
: If the first argument is '`!`', the expression is true if and only if the second argument is null. If the first argument is one of the listed unary operators, the expression is true if the unary test is true. If the first argument is not a legal unary operator, the expression is false.

3 arguments
: If the first argument is '`!`', the value is the negation of the two-argument test using the second and third arguments. If the second argument is one of the binary operators, the result of the expression is the result of the binary test using the first and third arguments as operands. If the first argument is exactly '`(`' and the third argument is exactly '`)`', the result is the one-argument test of the second argument. Otherwise, the expression is false. The '`-a`' and '`-o`' operators are considered binary operators in this case.

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 evaluated according to precedence. '`-a`' has a higher precedence than '`-o`'.

5 or more arguments
: The expression is parsed and evaluated according to precedence, with '`-a`' having a higher precedence than '`-o`'.

7.7 Bash Variables

These variables are set or used by Bash, but other shells do not normally treat them specially.

`BASH_ENV`
: If this variable is set when Bash is invoked to execute a shell script, its value is expanded and used as the name of a startup file to read before executing the script. See Section 7.2 [Bash Startup Files], page 51.

`TIMEFORMAT`
: The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the `time` reserved word should be displayed. The '`%`' character introduces an escape sequence that is expanded to a time value or other information. The escape sequences and their meanings are as follows; the 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 *MM*m*SS*.*FF*s. The value of *p* determines whether or not the fraction is included.

  If this variable is not set, bash acts as if it had the value

  `$'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'`.

  If the value is null, no timing information is displayed. A trailing newline is added when the format string is displayed.

`HISTCONTROL`
: Set to a value of '`ignorespace`', it means don't enter lines which begin with a space or tab into the history list. Set to a value of '`ignoredups`', it means don't enter lines which match the last entered line. A value of '`ignoreboth`' combines the two options. Unset, or set to any other value than those above, means to save all lines on the history list.

`HISTIGNORE`
: A colon-separated list of patterns used to decide which command lines should be saved on the history list. Each pattern is anchored at the beginning of the line and must fully specify the 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 characters, '`&`' matches the previous history line. '`&`' may be escaped using a backslash. The backslash is removed before attempting a match.

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

`HISTFILE`
: The name of the file to which the command history is saved. The default is '`~/.bash_history`'.

`HISTSIZE`
: If set, this is the maximum number of commands to remember in the history.

`HISTFILESIZE`
: The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines. The default value is 500. The history file is also truncated to this size after writing it when an interactive shell exits.

`histchars`
: Up to three characters which control history expansion, quick substitution, and tokenization (see Section 9.2 [History Interaction], page 82). The first character is the *history-expansion-char*, that is, the character which signifies the start of a history expansion, normally '`!`'. The second character is the character which signifies 'quick substitution' when seen as the first character on a line, normally '`^`'. The optional third character is the character which signifies the remainder of the line is a comment, when found as the first character of a word, usually '`#`'. The history comment character causes history substitution to be skipped for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

`HISTCMD`
: The history number, or index in the history list, of the current command. If `HISTCMD` is unset, it loses its special properties, even if it is subsequently reset.

`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. You can change the file interactively;

the next time you attempt to complete a hostname, Bash will add the contents of the new file to the already existing database.

`MAILCHECK`
: How often (in seconds) that the shell should check for mail in the files specified in `MAILPATH`.

`PROMPT_COMMAND`
: If present, this contains a string which is a command to execute before the printing of each primary prompt (`$PS1`).

`UID`
: The numeric real user id of the current user.

`EUID`
: The numeric effective user id of the current user.

`GROUPS`
: An array variable containing the list of groups of which the current user is a member.

`PPID`
: The process id of the shell's parent process.

`HOSTNAME`
: The name of the current host.

`HOSTTYPE`
: A string describing the machine Bash is running on.

`OSTYPE`
: A string describing the operating system Bash is running on.

`MACHTYPE`
: A string that fully describes the system type on which Bash is executing, in the standard GNU *cpu-company-system* format.

`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 Section 7.5 [The Set Builtin], page 60). 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 readonly.

`FIGNORE`
: A colon-separated list of suffixes to ignore when performing filename completion. A file name whose suffix matches one of the entries in `FIGNORE` is excluded from the list of matched file names. A sample value is '`.o:~`'

`GLOBIGNORE`
: A colon-separated list of patterns defining the set of filenames to be ignored by filename expansion. If a filename matched by a filename expansion pattern also matches one of the patterns in `GLOBIGNORE`, it is removed from the list of matches.

`DIRSTACK`
: An array variable (see Section 7.9 [Arrays], page 71) 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.

`PIPESTATUS`
: An array variable (see Section 7.9 [Arrays], page 71) containing a list of exit status values from the processes in the most-recently-executed foreground pipeline (which may contain only a single command).

`INPUTRC`
: The name of the Readline startup file, overriding the default of '`~/.inputrc`'.

`BASH`
: The full filename used to execute the current instance of Bash.

`BASH_VERSION`
: The version number of the current instance of Bash.

`BASH_VERSINFO`
: An 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`.

`SHLVL`
: Incremented by one each time a new instance of Bash is started. This is intended to be a count of how deeply your Bash shells are nested.

`OPTERR`
: If set to the value 1, Bash displays error messages generated by the `getopts` builtin command.

`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 filename expansion (see Section 3.10.5 [Filename Expansion], page 19).

`LC_MESSAGES`
: This variable determines the locale used to translate double-quoted strings preceded by a '`$`' (see Section 3.1.2.5 [Locale Translation], page 7).

`IGNOREEOF`
: Controls the action of the shell on receipt of an `EOF` character as the sole input. If set, then the value of it is the number of consecutive `EOF` characters that can be read as the first character on an input line before the shell will exit. If the variable exists but does not have a numeric value (or has no value) then the default is 10. If the variable does not exist, then `EOF` signifies the end of input to the shell. This is only in effect for interactive shells.

7.8 Shell Arithmetic

Bash includes several mechanisms to evaluate arithmetic expressions and display the result or use it as part of a command.

7.8.1 Arithmetic Evaluation

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by the `let` builtin.

Evaluation is done in long integers with no check for overflow, though division by 0 is trapped and flagged as an error. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

`- +`
: unary minus and plus

`! ~`
: logical and bitwise negation

`* / %`
: multiplication, division, remainder

`+ -`
: addition, subtraction

`<< >>`
: left and right bitwise shifts

`<= >= < >`
: comparison

`== !=`
: equality and inequality

`&`
: bitwise AND

`^` bitwise exclusive OR

`|` bitwise OR

`&&` logical AND

`||` logical OR

`expr ? expr : expr`
conditional evaluation

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

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

Constants with a leading 0 are interpreted as octal numbers. A leading '`0x`' or '`0X`' denotes hexadecimal. Otherwise, numbers take the form [*base*`#`]*n*, where *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base* is omitted, then base 10 is used. 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 interchangably 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.

7.8.2 Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

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

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

The evaluation is performed according to the rules listed above. If the expression is invalid, Bash prints a message indicating failure and no substitution occurs.

7.8.3 Arithmetic Builtins

`let`

`let` *expression* `[`*expression*`]`

The `let` builtin allows arithmetic to be performed on shell variables. Each *expression* is evaluated according to the rules given previously (see Section 7.8.1 [Arithmetic Evaluation], page 69). If the last *expression* evaluates to 0, `let` returns 1; otherwise 0 is returned.

7.9 Arrays

Bash provides one-dimensional array variables. Any variable may be used as an 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 contiguously. Arrays are zero-based.

An 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 greater than or equal to zero. To explicitly declare an array, use

`declare -a` *name*

The syntax

`declare -a` *name*`[`*subscript*`]`

is also accepted; the *subscript* is ignored. 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=(value`*1* `...` `value`*n*`)`

where each *value* is of the form `[[`*subscript*`]=]`*string*. If the optional subscript is 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. This syntax is also accepted by the `declare` builtin. Individual array elements may be assigned to using the `name[`*subscript*`]=`*value* syntax introduced above.

Any element of an array may be referenced using `${name[`*subscript*`]}`. The braces are required to avoid conflicts with the shell's filename expansion operators. If the *subscript* is '`@`' or '`*`', the word expands to all members of the array *name*. These subscripts differ only when the word appears within double quotes. If the word is double-quoted, `${name[*]}` expands to a single word with the value of each array member separated by the first character of the `IFS` variable, and `${name[@]}` expands each element of *name* to a separate word. When there are no array members, `${name[@]}` expands to nothing. This is analogous to the expansion of the special parameters '`@`' and '`*`'. `${#name[`*subscript*`]}` expands to the length of `${name[`*subscript*`]}`. If *subscript* is '`@`' or '`*`', the expansion is the number of elements in the array. Referencing an array variable without a subscript is equivalent to referencing element zero.

The `unset` builtin is used to destroy arrays. `unset` *name[subscript]* destroys the array element at index *subscript*. `unset` *name*, where *name* is an array, removes the entire array. A subscript of '`*`' or '`@`' also removes the entire array.

The `declare`, `local`, and `readonly` builtins each accept a '`-a`' option to specify an array. The `read` builtin accepts a '`-a`' option to assign a list of words read from the standard input to an array, and can read values from the standard input into individual array elements. The `set` and `declare` builtins display array values in a way that allows them to be reused as input.

7.10 Controlling the Prompt

The value of the variable `PROMPT_COMMAND` is examined just before Bash prints each primary prompt. If it is set and non-null, then the value is executed just as if you had typed it on the command line.

In addition, the following table describes the special characters which can appear in the prompt variables:

`\a` a bell character.

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

`\e` an escape character.

`\h` the hostname, up to the first '.'.

`\H` the hostname.

`\n` newline.

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

`\t` the time, in 24-hour HH:MM:SS format.

`\T` the time, in 12-hour HH:MM:SS format.

`\@` the time, in 12-hour am/pm format.

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

`\V` the release of Bash, version + patchlevel (e.g., 2.00.0)

`\w` the current working directory.

`\W` the basename of `$PWD`.

`\u` your username.

`\!` the history number of this command.

`\#` the command number of this command.

`\$` if the effective uid is 0, `#`, otherwise `$`.

`\nnn` the character corresponding to the octal number `nnn`.

`\\` a backslash.

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

`\]` end a sequence of non-printing characters.

7.11 The Restricted Shell

If Bash is started with the name `rbash`, or the '`--restricted`' 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. A restricted shell behaves identically to `bash` with the exception that the following are disallowed:

- Changing directories with the `cd` builtin.
- Setting or unsetting the values of the `SHELL` or `PATH` variables.
- Specifying command names containing slashes.
- Specifying a filename containing a slash as an argument to the `.` builtin command.

- Importing function definitions from the shell environment at startup.
- Redirecting output using the '>', '>|', '<>', '>&', '&>', and '>>' redirection operators.
- Using the `exec` builtin to replace the shell with another command.
- Adding or deleting builtin commands with the '-f' and '-d' options to the `enable` builtin.
- Specifying the '-p' option to the `command` builtin.
- Turning off restricted mode with '`set +r`'.

7.12 Bash POSIX Mode

Starting Bash with the '`--posix`' command-line option or executing '`set -o posix`' while Bash is running will cause Bash to conform more closely to the POSIX.2 standard by changing the behavior to match that specified by POSIX.2 in areas where the Bash default differs.

The following list is what's changed when 'POSIX mode' is in effect:

1. When a command in the hash table no longer exists, Bash will re-search `$PATH` to find the new location. This is also available with '`shopt -s checkhash`'.
2. The '>&' redirection does not redirect stdout and stderr.
3. The message printed by the job control code and builtins when a job exits with a non-zero status is 'Done(status)'.
4. Reserved words may not be aliased.
5. The POSIX.2 `PS1` and `PS2` expansions of '!' to the history number and '!!' to '!' are enabled, and parameter expansion is performed on the value regardless of the setting of the `promptvars` option.
6. Interactive comments are enabled by default. (Note that Bash has them on by default anyway.)
7. The POSIX.2 startup files are executed (`$ENV`) rather than the normal Bash files.
8. Tilde expansion is only performed on assignments preceding a command name, rather than on all assignment statements on the line.
9. The default history file is '`~/.sh_history`' (this is the default value of `$HISTFILE`).
10. The output of '`kill -l`' prints all the signal names on a single line, separated by spaces.
11. Non-interactive shells exit if *filename* in `.` *filename* is not found.
12. Redirection operators do not perform filename expansion on the word in the redirection unless the shell is interactive.
13. Function names must be valid shell `name`s. That is, they may not contain characters other than letters, digits, and underscores, and may not start with a digit. Declaring a function with an illegal name causes a fatal syntax error in non-interactive shells.

14. POSIX.2 'special' builtins are found before shell functions during command lookup.
15. If a POSIX.2 special builtin returns an error status, a non-interactive shell exits. The fatal errors are those listed in the POSIX.2 standard, and include things like passing incorrect options, redirection errors, variable assignment errors for assignments preceding the command name, and so on.
16. If the `cd` builtin finds a directory to change to using `$CDPATH`, the value it assigns to the `PWD` variable does not contain any symbolic links, as if '`cd -P`' had been executed.
17. A non-interactive shell exits with an error status if a variable assignment error occurs when no command name follows the assignment statements. A variable assignment error occurs, for example, when trying to assign a value to a read-only variable.
18. A non-interactive shell exits with an error status if the iteration variable in a `for` statement or the selection variable in a `select` statement is a read-only variable.
19. Process substitution is not available.
20. Assignment statements preceding POSIX.2 `special` builtins persist in the shell environment after the builtin completes.
21. The `export` and `readonly` builtin commands display their output in the format required by POSIX.2.

There is other POSIX.2 behavior that Bash does not implement. Specifically:

1. Assignment statements affect the execution environment of all builtins, not just special ones.

8 Job Control

This chapter disusses what job control is, how it works, and how Bash allows you to access its facilities.

8.1 Job Control Basics

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

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

```
[1] 25647
```

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

To facilitate the implementation of the user interface to job control, the system maintains the notion of a current terminal process group ID. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as `SIGINT`. These processes are said to be in the foreground. Background processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or write to the terminal. Background processes which attempt to read from (write to) the terminal are sent a `SIGTTIN` (`SIGTTOU`) signal by the terminal driver, which, unless caught, suspends the process.

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

There are a number of ways to refer to a job in the shell. The character '`%`' introduces a job name. Job number `n` may be referred to as '`%n`'. A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, '`%ce`' refers to a stopped `ce` job. Using '`%?ce`', on the other hand, refers to any job containing the string '`ce`' in its command line. If the prefix or substring matches more than one job, Bash reports an error. The symbols '`%%`' and '`%+`' refer to the shell's notion of the current job, which is the last job stopped while it was in the foreground. The previous job may be referenced using '`%-`'. In output pertaining to jobs (e.g., the output of the `jobs` command), the current job is always flagged with a '`+`', and the previous job with a '`-`'.

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

The shell learns immediately whenever a job changes state. Normally, Bash waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the the '`-b`' option to the `set` builtin is set, Bash reports such changes immediately (see Section 7.5 [The Set Builtin], page 60).

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

8.2 Job Control Builtins

`bg`

: `bg [`*jobspec*`]`

  Place *jobspec* into the background, as if it had been started with '`&`'. If *jobspec* is not supplied, the current job is used.

`fg`

: `fg [`*jobspec*`]`

  Bring *jobspec* into the foreground and make it the current job. If *jobspec* is not supplied, the current job is used.

`jobs`

```
jobs [-lpnrs] [jobspec]
jobs -x command [jobspec]
```

The first form lists the active jobs. The options have the following meanings:

`-l` List process IDs in addition to the normal information

`-n` Display information only about jobs that have changed status since you were last notified of their status.

`-p` List only the process ID of the job's process group leader.

`-r` Restrict output to running jobs.

`-s` Restrict output to stopped jobs.

If *jobspec* is given, output is restricted to information about that job. If *jobspec* is not supplied, the status of all jobs is listed.

If the '`-x`' option is supplied, `jobs` replaces any *jobspec* found in *command* or *arguments* with the corresponding process group ID, and executes *command*, passing it *arguments*, returning its exit status.

`kill`

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec
kill -l [sigspec]
```

Send a signal specified by *sigspec* or *signum* to the process named by *jobspec*. *sigspec* is either a signal name such as `SIGINT` (with or without the `SIG` prefix) or a signal number; *signum* is a signal number. If *sigspec* and *signum* are not present, `SIGTERM` is used. The '`-l`' option lists the signal names, or the signal name corresponding to *sigspec*.

`wait`

```
wait [jobspec|pid]
```

Wait until the child process specified by process ID *pid* or job specification *jobspec* exits and report its exit status. If a job spec is given, all processes in the job are waited for. If no arguments are given, all currently active child processes are waited for.

`disown`

```
disown [-h] [jobspec ...]
```

Without options, each *jobspec* is removed from the table of active jobs. If the '`-h`' option is given, the job is not removed from the table, but is marked so that `SIGHUP` is not sent to the job if the shell receives a `SIGHUP`. If *jobspec* is not present, the current job is used.

`suspend`

```
suspend [-f]
```

Suspend the execution of this shell until it receives a `SIGCONT` signal. The '`-f`' option means to suspend even if the shell is a login shell.

When job control is not active, the `kill` and `wait` builtins do not accept *jobspec* arguments. They must be supplied process IDs.

8.3 Job Control Variables

`auto_resume`

This variable controls how the shell interacts with the user and job control. If this variable exists then single word simple commands without redirects are treated as candidates for resumption of an existing job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, then the most recently accessed job will be selected. The name of a stopped job, in this context, is the command line used to start it. If this variable is set to the value '`exact`', the string supplied must match the name of a stopped job exactly; if set to '`substring`', the string supplied needs to match a substring of the name of a stopped job. The '`substring`' value provides functionality analogous to the '`%?`' job ID (see Section 8.1 [Job Control Basics], page 77). If set to any other value, the supplied string must be a prefix of a stopped job's name; this provides functionality analogous to the '`%`' job ID.

9 Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide. For information on using the GNU History Library in your own programs, see the GNU Readline Library Manual.

9.1 Bash History Facilities

When the '-o history' option to the set builtin is enabled (see Section 7.5 [The Set Builtin], page 60), the shell provides access to the *command history*, the list of commands previously typed. The text of the last HISTSIZE commands (default 500) is saved in a history list. The shell stores each command in the history list prior to parameter and variable expansion but after history expansion is performed, subject to the values of the shell variables HISTIGNORE and HISTCONTROL. When the shell starts up, the history is initialized from the file named by the HISTFILE variable (default '~/.bash_history'). HISTFILE is truncated, if necessary, to contain no more than the number of lines specified by the value of the HISTFILESIZE variable. When an interactive shell exits, the last HISTSIZE lines are copied from the history list to HISTFILE. If the histappend shell option is set (see Section 7.4 [Bash Builtins], page 53), the lines are appended to the history file, otherwise the history file is overwritten. If HISTFILE is unset, or if the history file is unwritable, the history is not saved. After saving the history, the history file is truncated to contain no more than $HISTFILESIZE lines. If HISTFILESIZE is not set, no truncation is performed.

The builtin command fc (see Section 6.2 [Korn Shell Builtins], page 46) may be used to list or edit and re-execute a portion of the history list. The history builtin (see Section 5.3 [C Shell Builtins], page 42) can be used to display or modify the history list and manipulate the history file. When using the command-line editing, search commands are available in each editing mode that provide access to the history list.

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 Section 7.4 [Bash Builtins], page 53 for a description of shopt.

9.2 Interactive History Expansion

The History library provides a history expansion feature that is similar to the history expansion provided by `csh`. This section describes the syntax used to manipulate the history information.

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 takes place in two parts. The first is to determine which line from the previous history should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is called the *event*, and the portions of that line that are acted upon are called *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion that Bash does, so that several English (or Unix) words surrounded by quotes are considered as one word. History expansions are introduced by the appearance of the history expansion character, which is '`!`' by default. Only '\' and '`'`' may be used to escape the history expansion character.

Several shell options settable with the `shopt` builtin (see Section 7.4 [Bash Builtins], page 53) may be used to tailor the behavior of history expansion. If the `histverify` shell option is enabled, and Readline is being used, history substitutions are not immediately passed to the shell parser. Instead, the expanded line is reloaded into the Readline editing buffer for further modification. If Readline is being used, and the `histreedit` shell option is enabled, a failed history expansion 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 with the `histchars` variable.

9.2.1 Event Designators

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

`!` Start a history substitution, except when followed by a space, tab, the end of the line, `=` or `(`.

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

`!-`*n* Refer to the command *n* lines back.

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

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

`!?`*string*`[?]`
: Refer to the most recent command containing *string*. The trailing '`?`' may be omitted if the *string* is followed immediately by a newline.

`^`*string1*`^`*string2*`^`
: Quick Substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to `!!:s/`*string1*`/`*string2*`/`.

`!#` The entire command line typed so far.

9.2.2 Word Designators

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

`0 (zero)` The `0`th word. For many applications, this is the command word.

n The *n*th word.

`^` The first argument; that is, word 1.

`$` The last argument.

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

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

`*` All of the words, except the `0`th. 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 a word designator is supplied without an event specification, the previous command is used as the event.

9.2.3 Modifiers

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

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

`t` Remove all leading pathname components, leaving the tail.

`r` Remove a trailing suffix of the form '.*suffix*', 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 spaces, tabs, and newlines.

`s/`*old*`/`*new*`/`
: Substitute *new* for the first occurrence of *old* in the event line. Any delimiter may be used in place of '`/`'. 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 '`&`'. The final delimiter is optional if it is the last character on the input line.

`&` Repeat the previous substitution.

`g` Cause changes to be applied over the entire event line. Used in conjunction with '`s`', as in `gs/`*old*`/`*new*`/`, or with '`&`'.

10 Command Line Editing

This chapter describes the basic features of the GNU command line editing interface.

10.1 Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text `C-K` is read as 'Control-K' and describes the character produced when the `K` key is pressed while the Control key is depressed.

The text `M-K` is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `K` key is pressed. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `K`. Either process is known as *metafying* the `K` key.

The text `M-C-K` is read as 'Meta-Control-k' and describes the character produced by *metafying* `C-K`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see Section 10.3 [Readline Init File], page 89).

10.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

10.2.1 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use your erase character to back up and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-B` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-F`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor are 'pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor are 'pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

`C-B` Move back one character.

`C-F` Move forward one character.

`DEL` Delete the character to the left of the cursor.

`C-D` Delete the character underneath the cursor.

Printing characters
Insert the character into the line at the cursor.

`C-_` Undo the last thing that you did. You can undo all the way back to an empty line.

10.2.2 Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-B`, `C-F`, `C-D`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-A` Move to the start of the line.

`C-E` Move to the end of the line.

`M-F` Move forward a word.

`M-B` Move backward a word.

`C-L` Clear the screen, reprinting the current line at the top.

Notice how `C-F` moves forward a character, while `M-F` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

10.2.3 Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* (re-inserting) it back into the line. If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it all. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

Here is the list of commands for killing text.

`C-K` Kill the text from the current cursor position to the end of the line.

`M-D` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL` Kill from the cursor the start of the previous word, or if between words, to the start of the previous word.

`C-W` Kill from the cursor to the previous whitespace. This is different than `M-DEL` because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

`C-Y` Yank the most recently killed text back into the buffer at the cursor.

`M-Y` Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `C-Y` or `M-Y`.

10.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-k`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' you type is a minus sign (`-`), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the `C-D` command an argument of 10, you could type '`M-1 0 C-d`'.

10.2.5 Searching for Commands in the History

Readline provides commands for searching through the command history (see Section 9.1 [Bash History Facilities], page 81) for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. The Escape character is used to terminate an incremental search. Control-J will also terminate the 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 terminate the search and accept the line, thereby executing the command from the history list.

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 part of the contents of the current line.

10.3 Readline Init File

Although the Readline library comes with a set of `emacs`-like keybindings installed by default, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an *inputrc* file in your home directory. The name of this file is taken from the value of the shell variable `INPUTRC`. If that variable is unset, the default is '`~/.inputrc`'.

When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.

In addition, the `C-x C-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

10.3.1 Readline Init File Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a '`#`' are comments. Lines beginning with a '`$`' indicate conditional constructs (see Section 10.3.2 [Conditional Init Constructs], page 92). Other lines denote variable settings and key bindings.

Variable Settings

You can change the state of a few variables in Readline by using the `set` command within the init file. Here is how you would specify that you wish to use `vi` line editing commands:

```
set editing-mode vi
```

Right now, there are only a few variables which can be set; so few, in fact, that we just list them here:

`bell-style`

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`' (the default), Readline attempts to ring the terminal's bell.

`comment-begin`

The string to insert at the beginning of the line when the `insert-comment` command is executed. The default value is `"#"`.

`completion-query-items`
: The number of possible completions that determines when the user is asked whether he wants to see the list of possibilities. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. The default limit is `100`.

`convert-meta`
: If set to 'on', Readline will convert characters with the eigth bit set to an ASCII key sequence by stripping the eigth bit and prepending an `ESC` character, converting them to a meta-prefixed key sequence. The default value is 'on'.

`disable-completion`
: 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`. The default is 'off'.

`editing-mode`
: The `editing-mode` variable controls which editing mode you are using. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either 'emacs' or 'vi'.

`enable-keypad`
: When set to 'on', readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. The default is 'off'.

`expand-tilde`
: If set to 'on', tilde expansion is performed when Readline attempts word completion. The default is 'off'.

`horizontal-scroll-mode`
: This variable can be set to either 'on' or 'off'. Setting it to 'on' means that the text of the lines that you edit will scroll horizontally on a single screen line when they are longer than the width of the screen, instead of wrapping onto a new screen line. By default, this variable is set to 'off'.

`keymap`
: Sets Readline's idea of the current keymap for key binding commands. Acceptable `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`mark-directories`
: If set to '`on`', completed directory names have a slash appended. The default is '`on`'.

`mark-modified-lines`
: This variable, when set to '`on`', says to display an asterisk ('`*`') at the start of history lines which have been modified. This variable is '`off`' by default.

`input-meta`
: If set to '`on`', Readline will enable eight-bit input (it will not strip the eighth bit from the characters it reads), regardless of what the terminal claims it can support. The default value is '`off`'. The name `meta-flag` is a synonym for this variable.

`output-meta`
: 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`'.

`show-all-if-ambiguous`
: 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. The default value is '`off`'.

`visible-stats`
: If set to '`on`', a character denoting a file's type is appended to the filename when listing possible completions. The default is '`off`'.

Key Bindings

The syntax for controlling key bindings in the init file is simple. First you have to know the name of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the init file. The name of the key can be expressed in different ways, depending on which is most comfortable for you.

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`, 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).

"*keyseq*": *function-name* or *macro*

keyseq differs from *keyname* above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

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

In the above example, '`C-u`' is bound to the function `universal-argument` (just as it was in the first example), '`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 following escape sequences are available when specifying key sequences:

`\C-` control prefix

`\M-` meta prefix

`\e` an escape character

`\\` backslash

`\"` "

`\'` '

When entering the text of a macro, single or double quotes should be used to indicate a macro definition. Unquoted text is assumed to be a function name. Backslash will quote any character in the macro text, including '"' and '''. For example, the following binding will make '`C-x \`' insert a single '`\`' into the line:

```
"\C-x\\": "\\"
```

10.3.2 Conditional Init 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 three parser directives used.

`$if` The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test extends to the end of the line; no characters are required to isolate it.

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

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

`application`
The *application* construct is used to include application-specific settings. Each program using the Readline library sets the *application name*, and you can test for it. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

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

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

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

10.3.3 Sample Init File

Here is an example of an inputrc file. This illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the Gnu Readline library.  Existing programs
# include FTP, Bash, and Gdb.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word Text after the function name is ignored

#
# Arrow keys in keypad mode
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Arrow keys in ANSI mode
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Arrow keys in 8 bit keypad mode
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Arrow keys in 8 bit ANSI mode
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history

C-q: quoted-insert

$endif
```

```
# An old-style binding.  This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word -- insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# insert a backslash (testing backslash escapes in sequences and macros)
"\C-x\\": "\\"
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather than converted to
# prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly rather than
# as meta-prefixed characters
set output-meta on

# if there are more than 150 possible completions for a word, ask the
# user if he wants to see all of them
set completion-query-items 150

# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

10.4 Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences.

10.4.1 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 letters and digits.

`backward-word (M-b)`
: Move back to the start of this, or the previous, word. Words are composed of letters and digits.

`clear-screen (C-l)`
: Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`
: Refresh the current line. By default, this is unbound.

10.4.2 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 setting of the `HISTCONTROL` variable. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)`
: Move 'up' through the history list.

`next-history (C-n)`
: Move 'down' through the history 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 you are entering.

`reverse-search-history (C-r)`
: Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`
: Search forward starting at the current line and moving 'down' through the the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`
: Search backward starting at the current line and moving 'up' through the history as necessary using a non-incremental search for a string supplied by the user.

`non-incremental-forward-search-history (M-n)`
: Search forward starting at the current line and moving 'down' through the the history as necessary using a non-incremental search for a string supplied by the user.

`history-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the current cursor position (the 'point'). This is a non-incremental search. By default, this command is unbound.

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

`yank-nth-arg (M-C-y)`
: Insert the first argument to the previous command (usually the second word on the previous line). 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 inserts the nth word from the end of the previous command.

`yank-last-arg (M-., M-_)`
: Insert last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like `yank-nth-arg`.

10.4.3 Commands For Changing Text

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

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)`
: Add the next character that you type to the line verbatim. This is how to insert key sequences like `C-Q`, for example.

`tab-insert (M-TAB)`
: Insert a tab character.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`transpose-chars (C-t)`
: Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. Negative argumentss don't work.

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

`upcase-word (M-u)`
: Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`downcase-word (M-l)`
: Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

`capitalize-word (M-c)`
: Capitalize the current (or following) word. With a negative argument, do the previous word, but do not move the cursor.

10.4.4 Killing And Yanking

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

`backward-kill-line (C-x Rubout)`
: Kill backward to the beginning of the line.

`unix-line-discard (C-u)`
: Kill backward from the cursor to the beginning of the current line. Save the killed text on the kill-ring.

`kill-whole-line ()`
: Kill all characters on the current line, no matter where the cursor is. By default, this is unbound.

`kill-word (M-d)`
: Kill from the cursor to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`
: Kill the word behind the cursor. Word boundaries are the same as `backward-word`.

`unix-word-rubout (C-w)`
: Kill the word behind the cursor, using white space as a word boundary. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`
: Delete all spaces and tabs around point. By default, this is unbound.

`kill-region ()`
: Kill the text between the point and the *mark* (saved cursor position. This text is referred to as the *region*. By default, this command is unbound.

`copy-region-as-kill ()`
: Copy the text in the region to the kill buffer, so you can yank it right away. By default, this command is unbound.

`copy-backward-word ()`
: Copy the word before point to the kill buffer. By default, this command is unbound.

`copy-forward-word ()`
: Copy the word following point to the kill buffer. By default, this command is unbound.

`yank (C-y)`
: Yank the top of the kill ring into the buffer at the current cursor position.

`yank-pop (M-y)`
: Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

10.4.5 Specifying Numeric Arguments

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

`universal-argument ()`
: This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

10.4.6 Letting Readline Type For You

`complete (TAB)`
: Attempt to do completion on the text before the cursor. This is application-specific. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion, and so on. 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 the cursor.

`insert-completions (M-*)`
: Insert all completions of the text before point that would have been generated by `possible-completions`.

`complete-filename (M-/)`
: Attempt filename completion on the text before point.

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

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

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

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

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

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

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

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

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

`dynamic-complete-history (M-TAB)`
: Attempt completion on the text before point, comparing the text against lines from the history list for possible completion matches.

`complete-into-braces (M-{)`
: Perform filename completion and return the list of possible completions enclosed within braces so the list is available to the shell (see Section 5.1 [Brace Expansion], page 41).

10.4.7 Keyboard Macros

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

`end-kbd-macro (C-x ))`
: Stop saving the characters typed into the current keyboard macro and save the definition.

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

10.4.8 Some Miscellaneous Commands

`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-uppercase-version (M-a, M-b, M-x, ...)`
: If the metafied character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

`prefix-meta (ESC)`
: Make the next character that you type be metafied. This is for people without a meta key. Typing '`ESC f`' is equivalent to typing '`M-f`'.

`undo (C-_, C-x C-u)`
: Incremental undo, separately remembered for each line.

`revert-line (M-r)`
: Undo all changes made to this line. This is like typing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-~)`
: Perform tilde expansion on the current word.

`set-mark (C-@)`
: Set the mark to the current 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 occurrences.

`character-search-backward (M-C-])`
: A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

`insert-comment (M-#)`
: The value of the `comment-begin` variable is inserted at the beginning of the current line, and the line is accepted as if a newline had been typed. This makes the current line a shell comment.

`dump-functions ()`
: Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`dump-variables ()`
: Print all of the settable 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. This command is unbound by default.

`dump-macros ()`
: Print all of the readline key sequences bound to macros and the strings they ouput. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`glob-expand-word (C-x *)`
: The word before point is treated as a pattern for pathname expansion, and the list of matching file names is inserted, replacing the word.

`glob-list-expansions (C-x g)`
: The list of expansions that would have been generated by `glob-expand-word` is inserted into the line, replacing the word before point.

`display-shell-version (C-x C-v)`
: Display version information about the current instance of Bash.

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

`history-expand-line (M-^)`
: Perform history expansion on the current line.

`alias-expand-line`
: Perform alias expansion on the current line (see Section 6.4 [Aliases], page 47).

`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. Any argument is ignored.

`emacs-editing-mode (C-e)`
: When in `vi` editing mode, this causes a switch back to `emacs` editing mode, as if the command '`set -o emacs`' had been executed.

10.5 Readline vi Mode

While the Readline library does not have a full set of `vi` editing functions, it does contain enough to allow simple editing of the line. The Readline `vi` mode behaves as specified in the POSIX 1003.2 standard.

In order to switch interactively between `emacs` and `vi` editing modes, use the '`set -o emacs`' and '`set -o vi`' commands (see Section 7.5 [The Set Builtin], page 60). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in 'insertion' mode, as if you had typed an '`i`'. Pressing `ESC` switches you into 'command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with '`k`' and subsequent lines with '`j`', and so forth.

11 Installing Bash

This chapter provides basic instructions for installing Bash on the various supported platforms. The distribution supports nearly every version of Unix (and, someday, GNU). Other independent ports exist for OS/2, Windows 95, and Windows NT.

11.1 Basic Installation

These are installation instructions for Bash.

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a '`Makefile`' in each directory of the package (the top directory, the '`builtins`' and '`doc`' directories, and the each directory under '`lib`'). It also creates a '`config.h`' file containing system-dependent definitions. Finally, it creates a shell script named `config.status` that you can run in the future to recreate the current configuration, a file '`config.cache`' that saves the results of its tests to speed up reconfiguring, and a file '`config.log`' containing compiler output (useful mainly for debugging `configure`). If at some point '`config.cache`' contains results you don't want to keep, you may remove or edit it.

If you need to do unusual things to compile the package, please try to figure out how `configure` could check whether or not to do them, and mail diffs or instructions to `bash-maintainers@prep.ai.mit.edu` so they can be considered for the next release.

The file '`configure.in`' is used to create `configure` by a program called Autoconf. You only need '`configure.in`' if you want to change it or regenerate `configure` using a newer version of Autoconf. If you do this, make sure you are using Autoconf version 2.10 or newer.

If you need to change '`configure.in`' or regenerate `configure`, you will need to create two files: '`_distribution`' and '`_patchlevel`'. '`_distribution`' should contain the major and minor version numbers of the Bash distribution, for example '`2.01`'. '`_patchlevel`' should contain the patch level of the Bash distribution, '`0`' for example. The script '`support/mkconffiles`' has been provided to automate the creation of these files.

The simplest way to compile Bash is:

1. `cd` to the directory containing the source code and type '`./configure`' to configure Bash for

your system. If you're using `csh` on an old version of System V, you might need to type '`sh ./configure`' instead to prevent `csh` from trying to execute `configure` itself.

Running `configure` takes awhile. While running, it prints some messages telling which features it is checking for.

2. Type '`make`' to compile Bash and build the `bashbug` bug reporting script.
3. Optionally, type '`make tests`' to run the Bash test suite.
4. Type '`make install`' to install `bash` and `bashbug`. This will also install the manual pages and Info file.

You can remove the program binaries and object files from the source code directory by typing '`make clean`'. To also remove the files that `configure` created (so you can compile Bash for a different kind of computer), type '`make distclean`'.

11.2 Compilers and Options

Some systems require unusual options for compilation or linking that the `configure` script does not know about. You can give `configure` initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

On systems that have the `env` program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

The configuration process uses GCC to build Bash if it is available.

11.3 Compiling For Multiple Architectures

You can compile Bash for more than one kind of computer at the same time, by placing the object files for each architecture in their own directory. To do this, you must use a version of `make` that supports the `VPATH` variable, such as GNU `make`. `cd` to the directory where you want the object files and executables to go and run the `configure` script from the source directory. You may need to supply the '`--srcdir=PATH`' argument to tell `configure` where the source files are. `configure` automatically checks for the source code in the directory that `configure` is in and in '..'.

If you have to use a make that does not supports the VPATH variable, you can compile Bash for one architecture at a time in the source code directory. After you have installed Bash for one architecture, use 'make distclean' before reconfiguring for another architecture.

Alternatively, if your system supports symbolic links, you can use the 'support/mkclone' script to create a build tree which has symbolic links back to each file in the source directory. Here's an example that creates a build directory in the current directory from a source directory '/usr/gnu/src/bash-2.0':

```
bash /usr/gnu/src/bash-2.0/support/mkclone -s /usr/gnu/src/bash-2.0 .
```

The mkclone script requires Bash, so you must have already built Bash for at least one architecture before you can create build directories for other architectures.

11.4 Installation Names

By default, 'make install' will install into '/usr/local/bin', '/usr/local/man', etc. You can specify an installation prefix other than '/usr/local' by giving configure the option '--prefix=PATH'.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you give configure the option '--exec-prefix=PATH', the package will use 'PATH' as the prefix for installing programs and libraries. Documentation and other data files will still use the regular prefix.

11.5 Specifying the System Type

There may be some features configure can not figure out automatically, but needs to determine by the type of host the package will run on. Usually configure can figure that out, but if it prints a message saying it can not guess the host type, give it the '--host=TYPE' option. 'TYPE' can either be a short name for the system type, such as 'sun4', or a canonical name with three fields: 'CPU-COMPANY-SYSTEM' (e.g., 'sparc-sun-sunos4.1.2').

See the file 'support/config.sub' for the possible values of each field.

11.6 Sharing Defaults

If you want to set default values for `configure` scripts to share, you can create a site shell script called `config.site` that gives default values for variables like `CC`, `cache_file`, and `prefix`. `configure` looks for '`PREFIX/share/config.site`' if it exists, then '`PREFIX/etc/config.site`' if it exists. Or, you can set the `CONFIG_SITE` environment variable to the location of the site script. A warning: the Bash `configure` looks for a site script, but not all `configure` scripts do.

11.7 Operation Controls

`configure` recognizes the following options to control how it operates.

`--cache-file=`*FILE*
: Use and save the results of the tests in *FILE* instead of '`./config.cache`'. Set *FILE* to '`/dev/null`' to disable caching, for debugging `configure`.

`--help`
: Print a summary of the options to `configure`, and exit.

`--quiet`
`--silent`
`-q`
: Do not print messages saying which checks are being made.

`--srcdir=`*DIR*
: Look for the Bash source code in directory *DIR*. Usually `configure` can determine that directory automatically.

`--version`
: Print the version of Autoconf used to generate the `configure` script, and exit.

`configure` also accepts some other, not widely used, boilerplate options.

11.8 Optional Features

The Bash `configure` has a number of '`--enable-`*FEATURE*' options, where *FEATURE* indicates an optional part of the package. There are also several '`--with-`*PACKAGE*' options, where *PACKAGE* is something like '`gnu-malloc`' or '`purify`' (for the Purify memory allocation checker). To turn off the default use of a package, use '`--without-`*PACKAGE*'. To configure Bash without a feature that is enabled by default, use '`--disable-`*FEATURE*'.

Here is a complete list of the '`--enable-`' and '`--with-`' options that the Bash `configure` recognizes.

`--with-afs`
: Define if you are using the Andrew File System from Transarc.

`--with-curses`
: Use the curses library instead of the termcap library. This should be supplied if your system has an inadequate or incomplete termcap database.

`--with-glibc-malloc`
: Use the GNU libc version of `malloc` in '`lib/malloc/gmalloc.c`'. This is somewhat slower than the default `malloc`, but wastes considerably less space.

`--with-gnu-malloc`
: Use the GNU version of `malloc` in '`lib/malloc/malloc.c`'. This is not the same `malloc` that appears in GNU libc, but an older version derived from the 4.2 BSD `malloc`. This `malloc` is very fast, but wastes a lot of space. This option is enabled by default. The '`NOTES`' file contains a list of systems for which this should be turned off, and `configure` disables this option automatically for a number of systems.

`--with-purify`
: Define this to use the Purify memory allocation checker from Pure Software.

`--enable-minimal-config`
: This produces a shell with minimal features, close to the historical Bourne shell.

The '`minimal-config`' option can be used to disable all of the following options, but it is processed first, so individual options may be enabled using '`enable-`*FEATURE*'.

All of the following options except for '`disabled-builtins`' and '`usg-echo-default`' are enabled by default, unless the operating system does not provide the necessary support.

`--enable-alias`
: Allow alias expansion and include the `alias` and `unalias` builtins.

`--enable-array-variables`
: Include support for one-dimensional array shell variables.

`--enable-bang-history`
: Include support for `csh`-like history substitution.

`--enable-brace-expansion`
: Include `csh`-like brace expansion (`b{a,b}c` $\mapsto$ `bac bbc`).

`--enable-command-timing`
: Include support for recognizing `time` as a reserved word and for displaying timing statistics for the pipeline following `time`. This allows pipelines as well as shell builtins and functions to be timed.

`--enable-directory-stack`
: Include support for a `csh`-like directory stack and the `pushd`, `popd`, and `dirs` builtins.

`--enable-disabled-builtins`
: Allow builtin commands to be invoked via '`builtin xxx`' even after `xxx` has been disabled using '`enable -n xxx`'. See Section 7.4 [Bash Builtins], page 53, for details of the `builtin` and `enable` builtin commands.

`--enable-dparen-arithmetic`
: Include support for the `ksh` `((...))` command.

`--enable-help-builtin`
: Include the `help` builtin, which displays help on shell builtins and variables.

`--enable-history`
: Include command history and the `fc` and `history` builtin commands.

`--enable-job-control`
: This enables job control features, if the OS supports them.

`--enable-process-substitution`
: This enables process substitution (see Section 3.10.3 [Process Substitution], page 18) if the OS provides the necessary support.

`--enable-prompt-string-decoding`
: Turn on the interpretation of a number of backslash-escaped characters in the `$PS1`, `$PS2`, `$PS3`, and `$PS4` prompt strings.

`--enable-readline`
: Include support for command-line editing and history with the Bash version of the Readline library.

`--enable-restricted`
: Include support for a *restricted shell*. If this is enabled, Bash, when called as `rbash`, enters a restricted mode. See Section 7.11 [The Restricted Shell], page 73, for a description of restricted mode.

`--enable-select`
: Include the `ksh` `select` builtin, which allows the generation of simple menus.

`--enable-usg-echo-default`
: Make the `echo` builtin expand backslash-escaped characters by default, without requiring the '`-e`' option. This makes the Bash `echo` behave more like the System V version.

The file 'config.h.top' contains C Preprocessor '#define' statements for options which are not settable from configure. Some of these are not meant to be changed; beware of the consequences if you do. Read the comments associated with each definition for more information about its effect.

Appendix A Reporting Bugs

Please report all bugs you find in Bash. But first, you should make sure that it really is a bug, and that it appears in the latest version of Bash that you have.

Once you have determined that a bug actually exists, use the `bashbug` command to submit a bug report. If you have a fix, you are encouraged to mail that as well! Suggestions and 'philosophical' bug reports may be mailed to `bug-bash@prep.ai.MIT.Edu` or posted to the Usenet newsgroup `gnu.bash.bug`.

All bug reports should include:

- The version number of Bash.
- The hardware and operating system.
- The compiler used to compile Bash.
- A description of the bug behaviour.
- A short script or 'recipe' which exercises the bug and may be used to reproduce it.

`bashbug` inserts the first three items automatically into the template it provides for filing a bug report.

Please send all reports concerning this manual to `chet@ins.CWRU.Edu`.

Appendix B Index of Shell Builtin Commands

Appendix C Shell Reserved Words

Appendix D Parameter and Variable Index

Appendix E Function Index

A

B

C

D

E

F

G

H

I

K

N

Appendix F Concept Index

M

N

O

P

Q

R

S

T

V

W

Y

Table of Contents