Bash Features

Overview Documentation for Bash
Edition 1.14, for `bash` Version 1.14.
August 1994

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

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

1.1 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.

1.2 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 an 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."
```

1.3 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 function named *name*. 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. The special parameter `#` that gives 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.

1.4 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.

`:` Do nothing beyond expanding any arguments and performing redirections.

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

`break` Exit from a `for`, `while`, or `until` loop.

`cd` Change the current working directory.

`continue` Resume the next iteration of an enclosing `for`, `while`, or `until` loop.

`echo` Print the arguments, separated by spaces, to the standard output.

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

`exec` If a *command* argument is supplied, it replaces the shell. If no *command* is specified, redirections may be used to affect the current shell environment.

`exit` Exit the shell.

`export` Mark the arguments as variables to be passed to child processes in the environment.

`getopts` Parse options to shell scripts or functions.

`hash` Remember the full pathnames of commands specified as arguments, so they need not be searched for on subsequent invocations.

`kill` Send a signal to a process.

`pwd` Print the current working directory.

`read` Read a line from the shell input and use it to set the values of specified variables.

`readonly` Mark variables as unchangable.

`return` Cause a shell function to exit with a specified value.

`shift` Shift positional parameters to the left.

`test`

`[` Evaluate a conditional expression.

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

`trap` Specify commands to be executed when the shell receives signals.

`umask` Set the shell process's file creation mask.

`unset` Cause shell variables to disappear.

`wait` Wait until child processes exit and report their exit status.

1.5 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.

`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.

`PS1` The primary prompt string.

`PS2` The secondary prompt string.

`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.

1.6 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.

1.6.1 Major Differences from the Bourne Shell

Bash implements the `!` keyword to negate the return value of a pipeline. Very useful when an `if` statement needs to act only if a test fails.

Bash includes brace expansion (see Section 2.2 [Brace Expansion], page 7).

Bash includes the Posix and `ksh`-style pattern removal `%%` and `##` constructs to remove leading or trailing substrings from variables.

The Posix and `ksh`-style `$()` form of command substitution is implemented, 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.

The expansion `${#xx}`, which returns the length of `$xx`, is supported.

The `IFS` variable is used to split only the results of expansion, not all words. 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, and thus useful recursive functions may be written.

The `noclobber` option is available to avoid overwriting existing files with output redirection.

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.

The `command` builtin allows selective disabling of functions when command lookup is performed.

Individual builtins may be enabled or disabled using the `enable` builtin.

Functions may be exported to children via the environment.

The Bash `read` builtin 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 `return` builtin may be used to abort execution of scripts executed with the `.` or `source` builtins.

The `umask` builtin allows symbolic mode arguments similar to those accepted by `chmod`.

The `test` builtin is slightly different, as it implements the Posix 1003.2 algorithm, which specifies the behavior based on the number of arguments.

2 C-Shell Style Features

The C-Shell (`csh`) was created by Bill Joy at UC 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 for controlling the interactive behaviour of the shell (e.g. `IGNOREEOF`).

See Chapter 6 [Using History Interactively], page 33 for details on history expansion.

2.1 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`'

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

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

2.2 Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *pathname expansion* (see the Bash manual page for details), 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,

```
a{d,c,b}e
```

expands into *ade ace abe*.

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

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

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

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

or

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

2.3 C Shell Builtins

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

`pushd`

`pushd [`*dir*` | +`*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`) 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`) to the top of the list by rotating the stack.

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]
```

Pops the directory stack, and `cd`s to the new top directory. When no arguments are given, removes the top directory from the stack and `cd`s 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.

`dirs`

```
dirs [+n | -n] [-l]
```

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.

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

`history`

```
history [n] [ [-w -r -a -n] [filename]]
```

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. Option `-w` means write out the current history to the history file; `-r` means to read the current history file and make its contents the history list. An argument of `-a` means to append the new history lines (history lines entered since the beginning of the current Bash session) to the history file. Finally, the `-n` argument means to read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current Bash session. If *filename* is given, then it is used as the history file, else if `$HISTFILE` has a value, that is used, otherwise '`~/.bash_history`' is used.

`logout` Exit a login shell.

`source` A synonym for `.` (see Section 1.4 [Bourne Shell Builtins], page 3)

2.4 C Shell Variables

`IGNOREEOF`

If this variable is set, it represents the number of consecutive `EOF`s Bash will read before exiting. By default, Bash will exit upon reading a single `EOF`.

`cdable_vars`

If this variable is set, Bash treats arguments to the `cd` command which are not directories as names of variables whose values are the directories to change to.

3 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.

3.1 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 the number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If `EOF` is read, the `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.

3.2 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 3.4 [Aliases], page 13).

`let` The `let` builtin allows arithmetic to be performed on shell variables. For details, refer to Section 4.7.3 [Arithmetic Builtins], page 26.

`typeset` The `typeset` command is supplied for compatibility with the Korn shell; however, it has been made obsolete by the `declare` command (see Section 4.4 [Bash Builtins], page 17).

3.3 Korn Shell Variables

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

`RANDOM` Each time this parameter is referenced, a random integer 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.

`PS4` This is the prompt printed before the command line is echoed when the `-x` option is set (see Section 4.5 [The Set Builtin], page 20).

`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.

3.4 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.

Aliases are not expanded when the shell is not interactive.

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

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

3.4.1 Alias Builtins

`alias`

    alias [name[=value] ...]

Without arguments, print the list of aliases on the standard output. 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.

4 Bash Specific Features

This section describes the features unique to Bash.

4.1 Invoking Bash

In addition to the single-character shell command-line options (see Section 4.5 [The Set Builtin], page 20), there are several multi-character options that you can use. These options must appear on the command line before the single-character options to be recognized.

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

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

`-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.

`-version` Display the version number of this shell.

`-login` Make this shell act as if it were directly invoked from login. This is equivalent to '`exec - 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`'.

`-nobraceexpansion`
: Do not perform curly brace expansion (see Section 2.2 [Brace Expansion], page 7).

`-nolineediting`
: Do not use the GNU Readline library (see Chapter 7 [Command Line Editing], page 37) to read interactive command lines.

`-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.

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.

`-i` Force the shell to run interactively.

`-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.

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.

4.2 Bash Startup Files

When and how Bash executes startup files.

```
For Login shells (subject to the -noprofile option):

    On logging in:
       If `/etc/profile' exists, then source it.

       If `~/.bash_profile' exists, then source it,
          else if `~/.bash_login' exists, then source it,
             else if `~/.profile' exists, then source it.

    On logging out:
       If `~/.bash_logout' exists, source it.

For non-login interactive shells (subject to the -norc and -rcfile options):
    On starting up:
       If `~/.bashrc' exists, then source it.

For non-interactive shells:
    On starting up:
       If the environment variable ENV is non-null, expand the
       variable and source the file named by the value.  If Bash is
       not started in Posix mode, it looks for BASH_ENV before
       ENV.
```

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

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

after (or before) any login specific initializations.

If Bash is invoked as `sh`, it tries to mimic the behavior of `sh` as closely as possible. For a login shell, it attempts to source only '`/etc/profile`' and '`~/.profile`', in that order. The `-noprofile` option may still be used to disable this behavior. A shell invoked as `sh` does not attempt to source any other startup files.

When Bash is started in *POSIX* mode, as with the `-posix` command line option, it follows the Posix 1003.2 standard for startup files. In this mode, the `ENV` variable is expanded and that file sourced; no other startup files are read.

4.3 Is This Shell Interactive?

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
```

You can ask an interactive Bash to not run your '`~/.bashrc`' file with the `-norc` flag. You can change the name of the '`~/.bashrc`' file to any other file name with `-rcfile` *filename*. You can ask Bash to not run your '`~/.bash_profile`' file with the `-noprofile` flag.

4.4 Bash Builtin Commands

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

`builtin`

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

  Run a shell builtin. This is useful when you wish to rename a shell builtin to be a function, but need the functionality of the builtin within the function itself.

`bind`

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

Display current Readline (see Chapter 7 [Command Line Editing], page 37) 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 7.3 [Readline Init File], page 40), but each binding must be passed as a separate argument: '`"\C-x\C-r":re-read-init-file`'. Options, if supplied, have the following meanings:

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

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

`-v`
: List current function names and bindings

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

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

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

`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 [-frxi] [`*name*`[=`*value*`]]`

Declare variables and/or give them attributes. If no *names* are given, then display the values of variables instead. `-f` means to use function names only. `-r` says to make *names* readonly. `-x` says to mark *names* for export. `-i` says that the variable is to be treated as an integer; arithmetic evaluation (see Section 4.7 [Shell Arithmetic], page 24) is performed when the variable is assigned a value. Using `+` instead of `-` turns off the attribute instead. When used in a function, `declare` makes *names* local, as with the `local` command.

`enable`

`enable [-n] [-a] [`*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 *names* become disabled.

Otherwise *names* are enabled. For example, to use the `test` binary found via `$PATH` instead of the shell builtin version, type '`enable -n test`'. The `-a` option means to list each builtin with an indication of whether or not it is enabled.

`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.

`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 `-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 [-acdmstfpnuvSH] [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 (the default if the `-H` option is not given).

`-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.

`-m` the maximum resident set size.

`-s` the maximum stack size.

`-t` the maximum amount of cpu time in seconds.

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

`-p` the pipe buffer size.

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

`-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 specified resource is printed. 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.

4.5 The Set Builtin

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

`set`

`set [-abefhkmnptuvxldCHP] [-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 command exits with a non-zero status.

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

`-h` Locate and remember (hash) commands as functions are defined, rather than when the function is executed.

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

`-m` Job control is enabled (see Chapter 5 [Job Control], page 29).

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

`-o` *option-name*

Set the flag corresponding to *option-name*:

`allexport`

same as `-a`.

`braceexpand`
: the shell will perform brace expansion (see Section 2.2 [Brace Expansion], page 7).

`emacs`
: use an emacs-style line editing interface (see Chapter 7 [Command Line Editing], page 37).

`errexit`
: same as `-e`.

`histexpand`
: same as `-H`.

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

`interactive-comments`
: allow a word beginning with a '`#`' to cause that word and all remaining characters on that line to be ignored in an interactive shell.

`monitor`
: same as `-m`.

`noclobber`
: same as `-C`.

`noexec`
: same as `-n`.

`noglob`
: same as `-f`.

`nohash`
: same as `-d`.

`notify`
: same as `-b`.

`nounset`
: same as `-u`.

`physical`
: same as `-P`.

`posix`
: change the behavior of Bash where the default operation differs from the Posix 1003.2 standard to match the standard. 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 `$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.

`-l` Save and restore the binding of the *name* in a `for` command.

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

`-C` Disallow output redirection to existing files.

`-H` Enable ! style history substitution. This flag is on by default.

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

`--` If no arguments follow this flag, then the positional parameters are unset. Otherwise, the positional parameters are set to the *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.

4.6 Bash Variables

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

`HISTCONTROL`
`history_control`

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.

`HISTFILE` The name of the file to which the command history is saved.

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

`histchars`
: Up to three characters which control history expansion, quick substitution, and tokenization (see Section 6.1 [History Interaction], page 33). 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.

`hostname_completion_file`
`HOSTFILE` Contains the name of a file in the same format as '`/etc/hosts`' that should be read when the shell needs to complete a hostname. 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.

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

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

`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:~`'

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

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

`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 characters 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.

`no_exit_on_failed_exec`
If this variable exists, the shell will not exit in the case that it couldn't execute the file specified in the `exec` command.

`nolinks` If present, says not to follow symbolic links when doing commands that change the current working directory. By default, bash follows the logical chain of directories when performing commands such as `cd` 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 `nolinks` exists, then:

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

See also the description of the `-P` option to the `set` builtin, Section 4.5 [The Set Builtin], page 20.

4.7 Shell Arithmetic

4.7.1 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

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

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

Constants with a leading 0 are interpreted as octal numbers. A leading `0x` or `0X` denotes hexadecimal. Otherwise, numbers take the form [*base#*]n, where *base* is a decimal number between 2 and 36 representing the arithmetic base, and *n* is a number in that base. If *base* is omitted, then base 10 is used.

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

4.7.2 Arithmetic Expansion

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

```
$[ expression ]
$(( expression ))
```

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

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

4.7.3 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 4.7.1 [Arithmetic Evaluation], page 24). If the last *expression* evaluates to 0, `let` returns 1; otherwise 0 is returned.

4.8 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 `PS1` variable:

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

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

`\n` newline.

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

`\w` the current working directory.

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

`\u` your username.

`\h` the hostname.

`\#` the command number of this command.

`\!` the history number of this command.

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

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

`\\` 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.

5 Job Control

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

5.1 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 4.5 [The Set Builtin], page 20). This feature is also controlled by the variable `notify`.

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

5.2 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 [-lpn] [`*jobspec*`]`
`jobs -x` *command* `[`*jobspec*`]`

The first form lists the active jobs. The `-l` option lists process IDs in addition to the normal information; the `-p` option lists only the process ID of the job's process group leader. The `-n` option displays only jobs that have changed status since last notfied. 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.

`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 active, the `kill` and `wait` builtins also accept *jobspec* arguments.

5.3 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 you have more than one job beginning with the string that you have 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 5.1 [Job Control Basics], page 29). 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.

`notify` Setting this variable to a value is equivalent to '`set -b`'; unsetting it is equivalent to '`set +b`' (see Section 4.5 [The Set Builtin], page 20).

6 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.

6.1 History Interaction

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

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*. 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.

6.1.1 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 `(`.

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

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

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

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

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

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

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

6.1.2 Word Designators

A : separates the event specification from the word designator. It can be omitted if the word designator begins with a `^`, `$`, `*` or `%`. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`0 (zero)`
: The `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.

6.1.3 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.

`r`
: Remove a trailing suffix of the form '`.`'*suffix*, leaving the basename.

`e`
: Remove all but the trailing suffix.

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

`p`
: Print the new command but do not execute it.

`q`
: Quote the substituted words, escaping further substitutions.

`x`
: Quote the substituted words as with `q`, but break into words at 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 `&`.

7 Command Line Editing

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

7.1 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 Control key is depressed and the `K` key is struck.

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 struck. 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 7.3 [Readline Init File], page 40, for more info).

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.2.4 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`.

7.3 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 *init* 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.

7.3.1 Readline Init 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 7.3.2 [Conditional Init Constructs], page 43). 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:

`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`.

`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`.

`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.

`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 `vi-comment` command is executed. The default value is `"#"`.

`meta-flag`
: 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`.

`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`.

`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`.

`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`.

`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-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command`; `emacs` is equivalent to `emacs-standard`. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`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`.

`expand-tilde`
: If set to `on`, tilde expansion is performed when Readline attempts word completion. 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\\": "\\"
```

7.3.2 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.

7.4 Bindable Readline Commands

7.4.1 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.

7.4.2 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 point. This is a non-incremental search. By default, this command is unbound.

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

`yank-nth-arg (M-C-y)`
: Insert the first argument to the previous command (usually the second word on the previous line). With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command.

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

7.4.3 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.

7.4.4 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.

`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.

7.4.5 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 ()`
: Each time this is executed, the argument count is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four. By default, this is not bound to a key.

7.4.6 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. See the Bash manual page for a complete list of available completion functions.

`possible-completions (M-?)`
: List the possible completions of the text before the cursor.

`insert-completions ()`
: Insert all completions of the text before point that would have been generated by `possible-completions`. By default, this is not bound to a key.

7.4.7 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.

7.4.8 Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
: Read in the contents of your init file, and incorporate any bindings or variable assignments found there.

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

`do-uppercase-version (M-a, M-b, ...)`
: Run the command that is bound to the corresoponding 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.

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

`display-shell-version (C-x C-v)`
: Display version information about the current instance of Bash.

`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.

`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.

7.5 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 4.5 [The Set Builtin], page 20). 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 following lines with '`j`', and so forth.

Appendix A Variable Index

T

U

Appendix B Concept Index

H

I

J

K

L

N

O

P

Q

R

S

T

U

W

Y

Table of Contents