The Z Shell Guide

Document Edition 2.1.10
15 October 1996

Original documentation by Paul Falstad

Document Edition 2.1.10, last updated 15 October 1996, of *The Z Shell Guide*, for zsh, Version 3.0.1.

This is a texinfo version of the man page for the Z Shell, originally by Paul Falstad. It was converted from the 'zsh.1' file distributed with zsh v2.5.0 by Jonathan Hardwick, jch@cs.cmu.edu and updated/modified by Clive Messer, clive@epos.demon.co.uk to its present state.

1 The Z Shell Guide

This document has been produced from the texinfo file '`zsh.texi`', included in the '`Doc`' subdirectory of the Zsh distribution.

1.1 Origins

The texinfo guide was originally put together by Jonathan Hardwick, `jch@cs.cmu.edu`, who converted the '`zsh.1`' file distributed with zsh v2.5.0. After a period of neglect it was updated by Clive Messer, `clive@epos.demon.co.uk` to reflect the many changes made to both the shell, the original '`zsh.1`', (which due to its size and ever increasing number of options has since been split into several man-pages: '`zsh.1`', '`zshall.1`', '`zshbuiltins.1`', '`zshcompctl.1`', '`zshcompctl.1`', '`zshexpn.1`', '`zshmisc.1`', '`zshoptions.1`', '`zshparam.1`', '`zshzle.1`'), and also now includes other useful information from the `META-FAQ`.

1.2 Producing documentation from zsh.texi.

Whilst this guide for the most part duplicates the man-pages, (suitably marked-up into texinfo), and is not intended to replace them, it does offer several advantages over them, not least that the texinfo source may be converted into several formats, e.g.,

The Info guide.
: The Info format allows searching for topics, commands, functions, etc. from the many Indices. The command `makeinfo zsh.texi` is used to produce the Info documentation.

The printed guide.
: The command `texi2dvi zsh.texi` will output '`zsh.dvi`' which can then be processed with `dvips` and optionally `gs` (Ghostscript) to produce a nicely formatted printed guide.

The html guide.
: Mark Borges, `mdb@cdc.noaa.gov`, maintains an html version of this guide at `http://www.mal.com/zsh/Doc/zsh_toc.html`.
(The html version is produced with `texi2html`, which may be obtained from `http://wwwcn.cern.ch/dci/texi2html/`. The command is, `texi2html -split_chapter -expandinfo zsh.texi`)

For those who do not have the necessary tools to process texinfo, precompiled documentation, (PostScript, dvi, info and html formats), is available from the zsh archive site or its mirrors in the file, '`zsh-doc.tar.gz`'. (See Section 2.2 [Availability], page 3, for a list of sites.)

1.3 Future

This guide is actively maintained by Clive Messer, `clive@epos.demon.co.uk`, and Mark Borges, `mdb@cdc.noaa.gov`. Patches, comments, criticism, and suggestions should be sent to either of the above or alternatively, the *zsh-workers* mailing list, `zsh-workers@math.gatech.edu`.

2 Introduction

Zsh is a UNIX command interpreter (shell) usable as an interactive login shell and as a shell script command processor. Of the standard shells, zsh most closely resembles ksh but includes many enhancements. Zsh has command line editing, builtin spelling correction, programmable command completion, shell functions (with autoloading), a history mechanism, and a host of other features.

2.1 Author

Zsh was originally written by Paul Falstad `pjf@cts.com`. Zsh is now maintained by the members of the zsh workers mailing list `zsh-workers@math.gatech.edu`. The development is currently coordinated by Zoltán Hidvégi, `hzoli@cs.elte.hu`.

2.2 Availability

Zsh is available from the following anonymous ftp sites. The first is the official archive site. The rest are mirror sites which are kept frequently up to date. The sites marked with `(G)` may mirror `ftp.math.gatech.edu` instead of the primary site.

Hungary	`ftp://ftp.cs.elte.hu/pub/zsh/` `ftp://ftp.kfki.hu/pub/packages/zsh/`
Australia	`ftp://ftp.ips.oz.au/pub/packages/zsh/ (G)`
France	`ftp://ftp.cenatls.cena.dgac.fr/pub/shells/zsh/`

Germany	`ftp://ftp.fu-berlin.de/pub/unix/shells/zsh/`
	`ftp://ftp.gmd.de/packages/zsh/`
	`ftp://ftp.uni-trier.de/pub/unix/shell/zsh/`

Japan	`ftp://ftp.tohoku.ac.jp/mirror/zsh/` `ftp://ftp.iij.ad.jp/pub/misc/zsh/`
Norway	`ftp://ftp.uit.no/pub/unix/shells/zsh/`
Sweden	`ftp://ftp.lysator.liu.se/pub/unix/zsh/`
UK	`ftp://ftp.net.lut.ac.uk/zsh/`

USA	`ftp://ftp.math.gatech.edu/pub/zsh/`
	`ftp://uiarchive.uiuc.edu/pub/packages/shells/zsh/`
	`ftp://ftp.sterling.com/zsh/ (G)`
	`ftp://ftp.rge.com/pub/shells/zsh/ (G)`

2.3 Undocumented Features

Known only to the recipients of the `zsh-workers` mailing list.
To join the mailing lists, see Section 2.4 [Mailing Lists], page 4.

2.4 Mailing Lists

Zsh has 3 mailing lists:

`zsh-announce@math.gatech.edu`
: Announcements about releases, major changes in the shell and the monthly posting of the Zsh FAQ. (moderated)

`zsh-users@math.gatech.edu`
: User discussions.

`zsh-workers@math.gatech.edu`
: Hacking, development, bug reports and patches.

To subscribe, send mail with the SUBJECT `subscribe <e-mail-address>` to the associated administrative address for the mailing list.

`zsh-announce-request@math.gatech.edu`
`zsh-users-request@math.gatech.edu`
`zsh-workers-request@math.gatech.edu`

YOU ONLY NEED TO JOIN ONE OF THE MAILING LISTS AS THEY ARE NESTED.

All submissions to `zsh-announce` are automatically forwarded to `zsh-users`.
All submissions to `zsh-users` are automatically forwarded to `zsh-workers`.

Un-subscribing is done similarly.

If you have problems subscribing/unsubscribing to any of the mailing lists, send mail to Richard Coleman, `coleman@math.gatech.edu`.

2.5 Further Information

2.5.1 The Zsh FAQ

Zsh has a list of Frequently Asked Questions (FAQ) maintained by Peter Stephenson, `P.Stephenson@swansea.ac.uk`. It is regularly posted to the newsgroup `comp.unix.shell` and the `zsh-announce` mailing list. The latest version can be found at any of the Zsh ftp sites, or at: `http://www.mal.com/zsh/FAQ/`

2.5.2 The Zsh Web Page

Zsh has a web page maintained by Mark Borges, `mdb@cdc.noaa.gov` which is located at: `http://www.mal.com/zsh/`.

2.5.3 See Also

sh(1), csh(1), tcsh(1), rc(1), bash(1), ksh(1), zshbuiltins(1), zshcompctl(1), zshexpn(1), zshparam(1), zshzle(1), zshoptions(1), zshmisc(1)

IEEE Standard for information Technology – Portable Operating System Interface (POSIX) – Part 2: Shell and Utilities, IEEE Inc, 1993, ISBN 1-55937-255-9.

3 Invocation

If the '`-s`' flag is not present and an argument is given, the first argument is taken to be the pathname of a script to execute. The remaining arguments are assigned to the positional parameters. The following flags are interpreted by the shell when invoked:

`-c` *string* Read commands from *string*.

`-i` Force shell to be interactive.

`-s` Read command from the standard input.

4 Startup/Shutdown Files

Commands are first read from '/etc/zshenv'. If the RCS option is unset within '/etc/zshenv', all other initialization files are skipped. Otherwise, commands are read from '$ZDOTDIR/.zshenv' (if ZDOTDIR is unset, HOME is used instead). If the first character of argument zero passed to the shell is -, or if the '-l' flag is present, then the shell is assumed to be a login shell, and commands are read from '/etc/zprofile' and then '$ZDOTDIR/.zprofile'. Then, if the shell is interactive and the NO_RCS option is unset, commands are read from '/etc/zshrc' and then '$ZDOTDIR/.zshrc'. Finally, if the shell is a login shell, '/etc/zlogin' and '$ZDOTDIR/.zlogin' are read.

4.1 Files

$ZDOTDIR/.zshenv
$ZDOTDIR/.zprofile
$ZDOTDIR/.zshrc
$ZDOTDIR/.zlogin
$ZDOTDIR/.zlogout
${TMPPREFIX}* (default is /tmp/zsh*)
/etc/zshenv
/etc/zprofile
/etc/zshrc
/etc/zlogin
/etc/zlogout

5 Shell Grammar

5.1 Simple Commands

A *simple command* is a sequence of optional parameter assignments followed by blank-separated words, with optional redirections interspersed. The first word is the command to be executed, and the remaining words, if any, are arguments to the command. If a command name is given, the parameter assignments modify the environment of the command when it is executed. The value of a simple command is its exit status, or 128 plus the signal number if terminated by a signal.

A *pipeline* is a sequence of one or more commands separated by `|` or `|&`. `|&` is shorthand for `2>&1 |`. The standard output of each command is connected to the standard input of the next command in the pipeline. If a pipeline is preceded by `coproc`, it is executed as a coprocess; a two-way pipe is established between it and the parent shell. The shell can read from or write to the coprocess by means of the `>&p` and `<&p` redirection operators. The value of a pipeline is the value of the last command. If a pipeline is not preceded by `!`, the value of that pipeline is the logical `NOT` of the value of the last command.

A *sublist* is a sequence of one or more pipelines separated by `&&` or `||`. If two pipelines are separated by `&&`, the second pipeline is executed only if the first is successful (returns a zero value). If two pipelines are separated by `||`, the second is executed only if the first is unsuccessful (returns a nonzero value). Both operators have equal precedence and are left associative.

A *list* is a sequence of zero or more sublists separated by, and optionally terminated by, `;`, `&`, `&|`, `&!`, or a newline. Normally the shell waits for each list to finish before executing the next one. If a list is terminated by a `&`, `&|`, or `&!`, the shell executes it in the background, and does not wait for it to finish.

5.2 Precommand Modifiers

A simple command may be preceded by a *precommand* modifier which will alter how the command is interpreted. These modifiers are shell builtin commands with the exception of `nocorrect` which is a reserved word.

`-`
: The command is executed with a `-` prepended to its `argv[0]` string.

`noglob`
: Filename generation (globbing) is not performed on any of the words.

`nocorrect`
: Spelling correction is not done on any of the words.

`exec`
: The command is executed in the parent shell without forking.

`command`
: The command word is taken to be the name of an external command, rather than a shell function or builtin.

5.3 Complex Commands

A *complex command* in zsh is one of the following:

`if` *list* `then` *list* `[ elif` *list* `then` *list* `] ... [ else` *list* `] fi`
: The `if` *list* is executed, and, if it returns a zero exit status, the `then` *list* is executed. Otherwise, the `elif` *list* is executed and, if its value is zero, the `then` *list* is executed. If each `elif` *list* returns nonzero, the `else` *list* is executed.

`for` *name* `[ in` *word* ... *term* `] do` *list* `done`
: Where *term* is one or more newline or `;`. Expand the list of *words*, and set the parameter *name* to each of them in turn, executing *list* each time. If the `in` *word* is omitted, use the positional parameters instead of the *words*.

`while` *list* `do` *list* `done`
: Execute the `do` *list* as long as the `while` *list* returns a zero exit status.

`until` *list* `do` *list* `done`
: Execute the `do` *list* as long as `until` *list* returns a nonzero exit status.

`repeat` *word* `do` *list* `done`
: *word* is expanded and treated as an arithmetic expression, which must evaluate to a number *n*. *list* is then executed *n* times.

`case` *word* `in [ [(]` *pattern* `[ |` *pattern* `] ... )` *list* `;; ] ... esac`
: Execute the *list* associated with the first *pattern* that matches *word*, if any. The form of the patterns is the same as that used for filename generation. See Section 6.7 [Filename Generation], page 20.

`select` *name* `[ in` *word* ... *term* `] do` *list* `done`
: Where *term* is one ore more newline or `;`. Print the set of *words*, each preceded by a number. If the `in` *word* is omitted, use the positional parameters. The `PROMPT3` prompt is printed and a line is read from standard input. If this line consists of the number of one of the listed *words*, then the parameter *name* is set to the *word* corresponding to this number. If this line is empty, the selection list is printed again. Otherwise, the value of the parameter *name* is set to null. The contents of the line read from standard input is saved in the parameter `REPLY`. *list* is executed for each selection until a break or end-of-file is encountered.

`(` *list* `)`
: Execute *list* in a subshell. Traps set by the `trap` builtin are reset to their default values while executing `list`.

`{` *list* `}`
: Execute *list*.

`function` *word* ... `[ () ] [` *term* `] {` *list* `}`
word ... `() [` *term* `] {` *list* `}`
word ... `() [` *term* `]` *command*
: Where *term* is one ore more newline or `;`. Define a function which is referenced by any one of *word*. Normally, only one *word* is provided; multiple *words* are usually only useful for setting traps. The body of the function is the *list* between the `{` and `}`. See Chapter 9 [Functions], page 33.

`time [` *pipeline* `]`
: The *pipeline* is executed, and timing statistics are reported on the standard error in the form specified by the `TIMEFMT` parameter. If *pipeline* is omitted, print statistics about the shell process and its children.

`[[` *exp* `]]`
: Evaluates the conditional expression *exp* and return a zero exit status if it is true. See Chapter 12 [Conditional Expressions], page 39, for a description of *exp*.

5.4 Alternate Forms For Complex Commands

Many of zsh's complex commands have alternate forms. These particular versions of complex commands should be considered deprecated and may be removed in the future. The versions in the previous section should be preferred instead. The short versions below only work if *sublist* is of the form `{` *list* `}` or if the `NO_SHORT_LOOPS` option is not set.

`if` *list* `{` *list* `} [ elif` *list* `{` *list* `} ]` ... `[ else {` *list* `} ]`
: An alternate form of `if`.

`if` *list sublist*
: A short form of previous one.

`for` *name* `(` *word* ... `)` *sublist*
: A short form of `for`.

`for` *name* `[ in` *word* ... *term* `]` *sublist*
: Where *term* is one or more newline or `;`. Another short form of `for`.

`foreach` *name* `(` *word* ... `)` *list* `end`
: Another form of `for`.

`while` *list* `{` *list* `}`
: An alternative form of `while`.

`until` *list* `{` *list* `}`
: An alternative form of `until`.

`repeat` *word sublist*
: This is a short form of `repeat`.

`case` *word* `{ [ [(]` *pattern* `[ |` *pattern* `]` ... `)` *list* `;; ]` ... `}`
: An alternative form of `case`.

`select` *name* `[ in` *word term* `]` *sublist*
: Where *term* is one or more newline or `;`. A short form of `select`.

5.5 Reserved Words

The following words are recognized as *reserved words* when used as the first word of a command unless quoted or disabled using `disable -r`:

`do done esac then elif else fi for case if while function repeat time until select coproc nocorrect foreach end ! [[ { }`

Additionally `}` is recognized in any position if the `IGNORE_BRACES` option is not set.

5.6 Comments

In non-interactive shells, or in interactive shells with the `INTERACTIVE_COMMENTS` option set, a word beginning with the third character of the `histchars` parameter (`#` by default) causes that word and all the following characters up to a newline to be ignored.

5.7 Aliasing

Every token in the shell input is checked to see if there is an alias defined for it. If so, it is replaced by the text of the alias if it is in command position (if it could be the first word of a simple command), or if the alias is global. If the text ends with a space, the next word in the shell input is treated as though it were in command position for purposes of alias expansion. An alias is defined using the `alias` builtin; global aliases may be defined using the '`-g`' option to that builtin.

Alias substitution is done on the shell input before any other substitution except history substitution. Therefore, if an alias is defined for the word '`foo`', alias substitution may be avoided by quoting part of the word, e.g. '`\foo`'. But there is nothing to prevent an alias being defined for '`\foo`' as well.

5.8 Quoting

A character may be *quoted* (that is, made to stand for itself) by preceding it with a `\`. `\` followed by a newline is ignored. All characters enclosed between a pair of single quotes (`''`) are quoted, except the first character of `histchars` (`!` by default). A single quote cannot appear within single quotes. Inside double quotes (`""`), parameter and command substitution occurs, and `\` quotes the characters `\`, `` ` ``, `"`, and `$`.

6 Expansion

The types of expansions performed are *history expansion*, *alias expansion*, *process substitution*, *parameter expansion*, *command substitution*, *arithmetic expansion*, *brace expansion*, *filename expansion*, and *filename generation*.

Expansion is done in the above specified order in five steps. The first is *history expansion* which is only performed in interactive shells. The next step is *alias expansion* which is done right before the command line is parsed. They are followed by *process substitution*, *parameter expansion*, *command substitution*, *arithmetic expansion*, and *brace expansion* which are performed in one step in left-to-right fashion. After these expansions, all unquoted occurrences of the characters `\`, `'`, and `"` are removed and the result is subjected to *filename expansion* followed by *filename generation*.

If the `SH_FILE_EXPANSION` option is set, the order of expansion is modified for compatibility with `sh` and `ksh`. *Filename expansion* is performed immediately after *alias substitution*, preceding the set of five substitutions mentioned above.

6.1 Filename Expansion

Each word is checked to see if it begins with an unquoted `~`. If it does, then the word up to a `/`, or the end of the word if there is no `/`, is checked to see if it can be substituted in one of the ways described here. If so, then the `~` and the checked portion are replaced with the appropriate substitute value.

A `~` by itself is replaced by the value of the `HOME` parameter. A `~` followed by a `+` or a `-` is replaced by the value of `PWD` or `OLDPWD`, respectively.

A `~` followed by a number is replaced by the directory at that position in the directory stack. `~0` is equivalent to `~+`, and `~1` is the top of the stack. `~+` followed by a number is replaced by the directory at that position in the directory stack. `~+0` is equivalent to `~+`, and `~+1` is the top of the stack. `~-` followed by a number is replaced by the directory that many positions from the bottom of the stack. `~-0` is the bottom of the stack. The `PUSHD_MINUS` option exchanges the effects of `~+` and `~-` where they are followed by a number.

A `~` followed by anything not already covered is looked up as a named directory, and replaced by the value of that named directory if found. Named directories are typically home directories for users on the system. They may also be defined if the text after the `~` is the name of a string shell parameter whose value begins with a `/`. It is also possible to define directory names using the '`-d`' option to the `hash` builtin.

In certain circumstances (in prompts, for instance), when the shell prints a path, the path is checked to see if it has a named directory as its prefix. If so, then the prefix portion is replaced with a `~` followed by the name of the directory. The shortest way of referring to the directory is used, with ties broken in favour of using a named directory, except when the directory is `/`.

If a word begins with an unquoted `=` and the `EQUALS` option is set, the remainder of the word is taken as the name of a command or alias. If a command exists by that name, the word is replaced by the full pathname of the command. If an alias exists by that name, the word is replaced with the text of the alias.

Filename expansion is performed on the right hand side of a parameter assignment, including those appearing after commands of the `typeset` family. In this case, the right hand side will be treated as a colon-separated list in the manner of `PATH` so that a `~` or an `=` following a `:` is eligible for expansion. All such behavior can be disabled by quoting the `~`, the `=`, or the whole expression (but not simply the colon); the `EQUALS` option is also respected.

If the option `MAGIC_EQUAL_SUBST` is set, any unquoted shell argument in the form *identifier*`=`*expression* becomes eligible for file expansion as described in the previous paragraph. Quoting the first `=` also inhibits this.

6.2 Process Substitution

Each command argument of the form `<(`*list*`)`, `>(`*list*`)` or `=(`*list*`)` is subject to process substitution. In the case of the `<` and `>` forms, the shell will run process *list* asynchronously, connected to a named pipe (FIFO). The name of this pipe will become the argument to the command. If the form with `>` is selected then writing to this file will provide input for *list*. If `<` is used, then the file passed as an argument will be a named pipe connected to the output of the *list* process. For example,

```
paste <(cut -f1 file1) <(cut -f3 file2) | tee >(process1) >(process2) >/dev/null
```

`cut`s fields 1 and 3 from the files *file1* and *file2* respectively, `paste`s the results together, and sends it to the processes *process1* and *process2*. Note that the file, which is passed as an argument to the command, is a system pipe so programs that expect to `lseek(2)` on the file will not work. Also note that the previous example can be more compactly and efficiently written as:

```
paste <(cut -f1 file1) <(cut -f3 file2) >>(process1) >>(process2)
```

The shell uses pipes instead of FIFOs to implement the latter two process substitutions in the above example.

If `=` is used, then the file passed as an argument will be the name of a temporary file containing the output of the *list* process. This may be used instead of the `<` form for a program that expects to `lseek(2)` on the input file.

6.3 Parameter Expansion

The character `$` is used to introduce parameter expansions. See Chapter 15 [Parameters], page 55, for a description of parameters. In the expansions discussed below that require a pattern, the form of the pattern is the same as that used for filename generation; See Section 6.7 [Filename Generation], page 20.

`${`*name*`}` The value, if any, of the parameter *name* is substituted. The braces are required if *name* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *name* is an array parameter, then the values of each element of *name* is substituted, one element per word. Otherwise, the expansion results in one word only; no word splitting is done on the result.

`${+`*name*`}` If *name* is the name of a set parameter, `1` is substituted, otherwise `0` is substituted.

`${`*name*`:-`*word*`}`
: If *name* is set and is non-null then substitute its value; otherwise substitute *word*. If *name* is missing, substitute *word*.

`${`*name*`:=`*word*`}`
: If *name* is unset or is null then set it to *word*; the value of the parameter is then substituted.

`${`*name*`::=`*word*`}`
: Set *name* to *word*; the value of the parameter is then substituted.

`${`*name*`:?`*word*`}`
: If *name* is set and is non-null, then substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, then a standard message is printed.

`${`*name*`:+`*word*`}`
: If *name* is set and is non-null then substitute *word*; otherwise substitute nothing.

`${`*name*`#`*pattern*`}`
`${`*name*`##`*pattern*`}`
: If the *pattern* matches the beginning of the value of *name*, then substitute the value of *name* with the matched portion deleted; otherwise, just substitute the value of *name*. In the first form, the smallest matching pattern is preferred; in the second form, the largest matching pattern is preferred. If name is an array and the substitution is not quoted or the `@` flag or the *name*`[@]` syntax is used, matching is performed on each array elements separately.

`${`*name*`%`*pattern*`}`
`${`*name*`%%`*pattern*`}`
: If the *pattern* matches the end of the value of *name*, then substitute the value of *name* with the matched portion deleted; otherwise, just substitute the value of *name*. In the first form, the smallest matching pattern is preferred; in the second form, the largest matching pattern is preferred. If name is an array and the substitution is not quoted or the `@` flag or the *name[@]* syntax is used, matching is performed on each array elements separately.

`${`*name*`:#`*pattern*`}`
: If the pattern matches the value of *name*, then substitute the empty string; otherwise, just substitute the value of *name*. If *name* is an array and the substitution is not quoted or the `@` flag or the *name*`[@]` syntax is used, matching is performed on each array elements separately, and the matched array elements are removed (use the `M` flag to remove the non-matched elements).

`${#`*spec*`}`
: If *spec* is one of the above substitutions, substitute the length in characters of the result instead of the result itself. If *spec* is an array expression, substitute the number of elements of the result.

`${^`*spec*`}`
: Turn on the value of the `RC_EXPAND_PARAM` option for the evaluation of *spec*; if the `^` is doubled, turn it off. When this option is set, array expansions of the form '`foo${xx}bar`', where the parameter '`xx`' is set to '`(a b c)`', are substituted with '`fooabar foobbar foocbar`' instead of the default '`fooa b cbar`'.

`${=`*spec*`}`
: Turn on the value of the `SH_WORD_SPLIT` option for the evaluation of *spec*; if the `=` is doubled, turn it off. When this option is set, parameter values are split into separate words using `IFS` as a delimiter before substitution. This is done by default in most other shells.

`${~`*spec*`}`
: Turn on the value of the `GLOB_SUBST` option for the evaluation of *spec*; if the `~` is doubled, turn it off. When this option is set, any pattern characters resulting from the substitution become eligible for file expansion and filename generation.

If the colon is omitted from one of the above expressions containing a colon, then the shell only checks whether *name* is set or not, not whether it is null.

If a `${...}` type parameter expression or a `$(...)` type command substitution is used in place of *name* above, it is substituted first and the result is used as it were the value of *name*.

If the opening brace is directly followed by an opening parenthesis the string up to the matching closing parenthesis will be taken as a list of flags. Where arguments are valid, any character, or the matching pairs `(...)`, `{...}`, `[...]`, or `<...>`, may be used in place of the colon as delimiters. The following flags are supported:

`A` Create an array parameter with `${...:=...}` or `${...::=...}`. Assignment is made before sorting or padding.

`@` In double quotes, array elements are put into separate words. Eg. `${(@)`*foo*`}` is equivalent to `${`*foo*`[@]}` and `${(@)`*foo*`[1,2]}` is the same as `$`*foo*`[1] $`*foo*`[2]`.

`e` Perform *parameter expansion*, *command substitution* and *arithmetic expansion* on the result. Such expansions can be nested but too deep recursion may have unpredictable effects.

`o` Sort the resulting words in ascending order.

`O` Sort the resulting words in descending order.

`i` With `o` or `O`, makes the sort case-insensitive.

`L` Converts all letters in the result to lowercase.

`U` Converts all letters in the result to uppercase.

`C` Capitalizes the resulting words

`c` With `${#`*name*`}`, count the total number of characters in an array, as if the elements were concatenated with spaces between them.

`w` With `${#`*name*`}`, count words in arrays or strings; the `s` flag may be used to set a word delimiter.

`W` Similar to `w` with the difference that empty words between repeated delimiters are also counted.

`p` Recognize the same escape sequences as the `print` builtin in string arguments to subsequent flags.

`l:`*expr*`::`*string1*`::`*string2*`:`
Pad the resulting words on the left. Each word will be truncated if required and placed in a field *expr* characters wide. The space to the left will be filled with *string1* (concatenated as often as needed), or spaces if *string1* is not given. If both *string1* and *string2* are given, this string will be placed exactly once directly to the left of the resulting word.

`r:`*expr*`::`*string1*`::`*string2*`:`
As `l`, but pad the words on the right.

`j:`*string*`:` Join the words or arrays together using *string* as a separator. Note that this occurs before word splitting by the `SH_WORD_SPLIT` option.

`F` Join the words of arrays together using newline as a separator. This is a shorthand for `pj:\n:`.

`s:`*string*`:` Force word splitting (see the option `SH_WORD_SPLIT`) at the separator *string*. Splitting only occurs in places where an array value is valid.

`f` Split the result of the expansion to lines. This is a shorthand for `ps:\n:`.

`S` (This and all remaining flags are used with the `${...#...}` and `${...%...}` forms). Search substrings as well as beginnings or ends.

`I:`*expr*`:` Search the *expr*'th match (where *expr* evaluates to a number).

`M` Include the matched portion in the result.

`R` Include the unmatched portion in the result (the *R*est).

`B` Include the index of the beginning of the match in the result.

`E` Include the index of the end of the match in the result.

`N` Include the length of the match in the result.

6.4 Command Substitution

A command enclosed in parentheses preceded by a dollar sign, like so: `$(...)` or quoted with grave accents: `` `...` `` is replaced with its standard output, with any trailing newlines deleted. If the substitution is not enclosed in double quotes, the output is broken into words using the `IFS` parameter. The substitution `$(cat` *foo*`)` may be replaced by the equivalent but faster `$(<`*foo*`)`. In either case, if the option `GLOB_SUBST` is set the output is eligible for filename generation.

6.5 Arithmetic Expansion

A string of the form `$[`*exp*`]` is substituted with the value of the arithmetic expression *exp*. *exp* is subjected to *parameter expansion*, *command substitution* and *arithmetic expansion* before it is evaluated. See Chapter 11 [Arithmetic Evaluation], page 37.

6.6 Brace Expansion

A string of the form '`foo{xx,yy,zz}bar`' is expanded to the individual words '`fooxxbar`', '`fooyybar`', and '`foozzbar`'. Left-to-right order is preserved. This construct may be nested. Commas may be quoted in order to include them literally in a word.

An expression of the form `{`*n1*`..`*n2*`}`, where *n1* and *n2* are integers, is expanded to every number between *n1* and *n2*, inclusive. If either number begins with a zero, all the resulting numbers will be padded with leading zeroes to that minimum width. If the numbers are in decreasing order the resulting sequence will also be in decreasing order.

If a brace expression matches none of the above forms, it is left unchanged, unless the `BRACE_CCL` option is set. In that case, it is expanded to a sorted list of the individual characters between the braces, in the manner of a search set. `-` is treated specially as in a search set, but `^` or `!` as the first character is treated normally.

6.7 Filename Generation

If a word contains an unquoted instance of one of the characters `*`, `|`, `<`, `[`, or `?`, it is regarded as a pattern for filename generation, unless the `GLOB` option is unset. If the `EXTENDED_GLOB` option is set, the `^`, `~`, and `#` characters also denote a pattern; otherwise (except for an initial `~`, See Section 6.1 [Filename Expansion], page 15) they are not treated specially by the shell. The word is replaced with a list of sorted filenames that match the pattern. If no matching pattern is found, the shell gives an error message, unless the `NULL_GLOB` option is set, in which case the word is deleted; or unless the `NOMATCH` option is unset, in which case the word is left unchanged. In filename generation, the character `/` must be matched explicitly; also, a `.` must be matched explicitly at the beginning of a pattern or after a `/`, unless the `GLOB_DOTS` option is set. No filename generation pattern matches the files '`.`' or '`..`'. In other instances of pattern matching, the `/` and `.` are not treated specially.

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

`?` Matches any character.

`[...]` Matches any of the enclosed characters. Ranges of characters can be specified by separating two characters by a `-`. A `-` or `]` may be matched by including it as the first character in the list.

`[^...]`
`[!...]` Like `[...]`, except that it matches any character which is not in the given set.

`<x-y>` Matches any number in the range `x` to `y`, inclusive. If `x` is omitted, the number must be less than or equal to `y`. If `y` is omitted, the number must be greater than or equal to `x`. A pattern of the form `<->` matches any number.

`^x` Matches anything except the pattern `x`.

`x|y` Matches either `x` or `y`.

`x#` Matches zero or more occurrences of the pattern `x`.

`x##` Matches one or more occurrences of the pattern `x`.

Parentheses may be used for grouping. Note that the `|` character must be within parentheses, so that the lexical analyzer does not think it is a pipe character. Also note that `/` has a higher precedence than `^`; that is:

```
ls ^foo/bar
```

will search directories in '`.`' except '`./foo`' for a file named '`bar`'.

A pathname component of the form `(`*foo*`/)#` matches a path consisting of zero or more directories matching the pattern *foo*. As a shorthand, `**/` is equivalent to `(*/)#`. Thus:

```
ls (*/)#bar
```

or

```
ls **/bar
```

does a recursive directory search for files named *bar*, not following symbolic links. For this you can use the form `***/`.

If used for filename generation, a pattern may contain an exclusion specifier. Such patterns are of the form *pat1*`~`*pat2*. This pattern will generate all files matching *pat1*, but which do not match *pat2*. For example, '`*.c~lex.c`' will match all files ending in '`.c`', except the file '`lex.c`'. This may appear inside parentheses. Note that `~` has higher precedence than `|`, so that '`pat1|pat2~pat3`' matches any time that *pat1* matches, or if *pat2* matches while *pat3* does not. Note also that any `/` characters are not treated specially in the exclusion specifier, so that a `*` will match multiple path segments if they appear in the pattern to the left of the `~`.

Patterns used for filename generation may also end in a list of qualifiers enclosed in parentheses. The qualifiers specify which filenames that otherwise match the given pattern will be inserted in the argument list. A qualifier may be any one of the following:

`/` Directories

`.` Plain files

`@` Symbolic links

`=` Sockets

`p` Named pipes (FIFOs)

`*` Executable plain files (0100)

`%` Device files (character or block special)

`%b` Block special files

`%c` Character special files

`r` owner-readable files (0400)

`w` owner-writable files (0200)

`x` owner-executable files (0100)

`A` group-readable files (0040)

`I` group-writable files (0020)

`E` group-executable files (0010)

`R` world-readable files (0004)

`W` world-writable files (0002)

`X` world-executable files (0001)

`s` Setuid files (04000)

`S` Setgid files (02000)

`t` files with the sticky bit (01000)

`d`*dev* Files on the device *dev*

`l[+|-]`*ct* Files having a link count less than *ct* (`-`), greater than *ct* (`+`), or is equal to *ct*.

`U` Files owned by the effective user id.

`G` Files owned by the effective group id.

`u`*id* Files owned by user *id* if *id* is a number. If not, the character after the `u` will be used as a separator and the string between it and the next matching separator (`(`, `[`, `{`, and `<` match `)`, `]`, `}`, and `>` respectively; any other character matches itself) will be taken as a user name and translated into the corresponding user id (e.g. `u:foo:` or `u[foo]` for user `foo`).

gid Like u*id* but with group ids or names.

a[Mwhm][-|+]*n*
: Files accessed exactly *n* days ago. Files accessed within the last *n* days are selected using a negative value for *n* ('-n'). Files accessed more than *n* days ago are selected by a positive *n* value (+n). Optional unit specifiers M, w, h, or m (e.g. ah5) cause the check to be performed with months (of 30 days), weeks, hours, or minutes instead of days, respectively. For instance, `echo *(ah-5)` would echo files accessed within the last five hours.

m[Mwhm][-|+]*n*
: Like the file access qualifier, except that it uses the file modification time.

c[Mwhm][-|+]*n*
: Like the file access qualifier, except that it uses the file inode change time.

L[+-]*n* Files less than *n* bytes (-), more than *n* bytes (+), or exactly *n* bytes in length. If this flag is directly followed by a k (K), m (M), or p (P) (e.g. Lk+50) the check is performed with kilobytes, megabytes, or blocks (of 512 bytes) instead.

^ Negates all qualifiers following it.

- Toggles between making the qualifiers work on symbolic links (the default), and the files they point to.

M Sets the MARK_DIRS option for the current pattern.

T Appends a trailing qualifier mark to the file names, analogous to the LIST_TYPES, for the current pattern (overrides M).

N Sets the NULL_GLOB option for the current pattern.

D Sets the GLOB_DOTS option for the current pattern.

More than one of these lists can be combined, separated by commas; the whole list matches if at least one of the sublists matches (they are or'ed, the qualifiers in the sublists are and'ed).

If a : appears in a qualifier list, the remainder of the expression in parentheses is interpreted as a modifier (See Section 6.8.3 [Modifiers], page 24). Note that each modifier must be introduced by a separate :. Note also that the result after modification does not have to be an existing file. The name of any existing file can be followed by a modifier of the form (:...) even if no filename generation is performed.

Thus:

```
ls *(-/)
```

lists all directories and symbolic links that point to directories, and

```
ls *(%W)
```

lists all world-writable device files in the current directory, and

```
ls *(W,X)
```

lists all files in the current directory that are world-writable or world-executable, and

```
echo /tmp/foo*(u0^@:t)
```

outputs the basename of all root-owned files beginning with the string `foo` in '`/tmp`', ignoring symlinks, and

```
ls *.*~(lex|parse).[ch](^D^l1)
```

lists all files having a link count of one whose names contain a dot (but not those starting with a dot, since `GLOB_DOTS` is explicitly switched off) except for '`lex.c`', '`lex.h`', '`parse.c`', and '`parse.h`'.

6.8 History Expansion

History substitution allows you to use words from previous command lines in the command line you are typing. This simplifies spelling corrections and the repetition of complicated commands or arguments. Command lines are saved in the history list, the size of which is controlled by the `HISTSIZE` variable. The most recent command is retained in any case. A history substitution begins with the first character of the `histchars` parameter which is `!` by default and may occur anywhere on the command line; history substitutions do not nest. The `!` can be escaped with `\` or can be enclosed between a pair of single quotes (`''`) to suppress its special meaning. Double quotes will not work for this.

Input lines containing history substitutions are echoed on the terminal after being expanded, but before any other substitutions take place or the command gets executed.

6.8.1 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 blank, newline, `=`, or `(`.

`!!` Refer to the previous command. By itself, this substitution repeats the previous command.

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

`!-`*n* Refer to the current command-line minus *n*.

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

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

`!#` Refer to the current command line typed in so far. The line is treated as if it were complete up to and including the word before the one with the `!#` reference.

`!{...}` Insulate a history reference from adjacent characters (if necessary).

6.8.2 Word Designators

A word designator indicates which word or words of a given command line will be included in a history reference. A `:` separates the event specification from the word designator. It can be omitted if the word designator begins with a `^`, `$`, `*`, `-` or `%`. Word designators include:

`0` The first input word (command).

n The *n*'th argument.

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

`$` The last argument.

`%` The word matched by (the most recent) `?`*str* search.

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

`*` All the arguments, or a null value if there is just one word in the event.

`x*` Abbreviates `x-$`.

`x-` Like `x*` but omitting word `$`.

Note that a `%` word designator will only work when used as `!%`, `!:%`, `!?str?:%` and only when used after a `!?` substitution. Anything else will result in an error, although the error may not be the most obvious one.

6.8.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`. These modifiers also work on the result of filename and parameter expansion.

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

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

`e` Remove all but the suffix.

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

`&` Repeat the previous substitution.

`g` Apply the change to the first occurrence of a match in each word, by prefixing the above (for example, `g&`).

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

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

`x` Like `q`, but break into words at each blank.

`l` Convert the words to all lowercase.

`u` Convert the words to all uppercase.

`f` Repeats the immediately (without a colon) following modifier until the resulting word doesn't change any more. This and the following `F`, `w` and `W` modifier only work with parameter and filename expansion.

`F:`*expr*`:` Like `f`, but repeats only `n` times if the expression *expr* evaluates to `n`. Any character can be used instead of the `:`, if any of `(`, `[`, or `{` is used as the opening delimiter the second one has to be `)`, `]`, or `}` respectively.

`w` Makes the immediately following modifier work on each word in the string.

`W:`*sep*`:` Like `w` but words are considered to be the parts of the string that are separated by *sep*. Any character can be used instead of the `:`, opening parentheses are handled specially, see above.

`s/`*l*`/`*r*`[/]` Substitute *r* for *l*.

Unless preceded by a `g`, the substitution is done only for the first string that matches `l`.

The left-hand side of substitutions are not regular expressions, but character strings. Any character can be used as the delimiter in place of `/`. A backslash quotes the delimiter character. The character `&`, in the right hand side, is replaced by the text from the left-hand-side. The `&` can be quoted with a backslash. A null `l` uses the previous string either from a `l` or from a contextual scan string `s` from `!?s`. You can omit the rightmost delimiter if a newline immediately follows `r`; the right-most `?` in a context scan can similarly be omitted.

By default, a history reference with no event specification refers to the same line as the last history reference on that command line, unless it is the first history reference in a command. In that case, a history reference with no event specification always refers to the previous command. However, if the option `CSH_JUNKIE_HISTORY` is set, then history reference with no event specification will always refer to the previous command. For example, `!!:1` will always refer to the first word of the previous command and `!!$` will always refer to the last word of the previous command. And with `CSH_JUNKIE_HISTORY` set, then `!:1` and `!$` will function in the same manner as `!!:1` and `!!$`, respectively. However, if `CSH_JUNKIE_HISTORY` is unset, then `!:1` and `!$` will refer to the first and last words respectively, of the last command referenced on the current command line. However, if they are the first history reference on the command line, then they refer to the previous command.

The character sequence *^foo^bar* repeats the last command, replacing the string *foo* with the string *bar*.

If the shell encounters the character sequence `!"` in the input, the history mechanism is temporarily disabled until the current list is fully parsed. The `!"` is removed from the input, and any subsequent `!` characters have no special significance.

A less convenient but more comprehensible form of command history support is provided by the `fc` builtin (see Chapter 17 [Shell Builtin Commands], page 75).

7 Redirection

Before a command is executed, its input and output may be redirected. The following may appear anywhere in a simple command or may precede or follow a complex command. Substitution occurs before *word* or *digit* is used except as noted below. If the result of substitution on *word* produces more than one filename, redirection occurs for each separate filename in turn.

`<`*word*
: Open file *word* as standard input.

`<>`*word*
: Open file *word* for reading and writing as standard input. If the file does not exist then it is created.

`>`*word*
: Open file *word* as standard output. If the file does not exist then it is created. If the file exists, and the `CLOBBER` option is unset, this causes an error; otherwise, it is truncated to zero length.

`>|` *word*
`>!` *word*
: Same as `>`, except that the file is truncated to zero length if it exists, even if `CLOBBER` is unset.

`>>`*word*
: Open file *word* as standard output. If the file exists then output is appended to it. If the file does not exist, and the `CLOBBER` option is unset, this causes an error; otherwise, the file is created.

`>>|` *word*
`>>!` *word*
: Same as `>>`, except that the file is created if it does not exist, even if `CLOBBER` is unset.

`<<[-]` *word*
: The shell input is read up to a line that is the same as *word*, or to an end-of-file. No parameter substitution, command substitution or filename generation is performed on *word*. The resulting document, called a *here-document*, becomes the standard input. If any character of *word* is quoted with single or double quotes or a `\`, no interpretation is placed upon the characters of the document. Otherwise, parameter and command substitution occurs, `\` followed by a newline is removed, and `\` must be used to quote the characters `\`, `$`, `` ` ``, and the first character of *word*. If `<<-` is used, then all leading tabs are stripped from *word* and from the document.

`<<<`*word*
: Perform shell expansion on *word* and pass the result to standard input.

`<&`*digit*
: The standard input is duplicated from file descriptor *digit* (see `dup(2)`). Similarly for standard output using `>&`*digit*.

`>&`*word*
: Same as `>`*word* `2>&1`.

`>>&`*word*
: Same as `>>`*word* `2>&1`.

`<&-`
: Close the standard input.

`>&-`
: Close the standard output.

`<&p`
: The input from the coprocess is moved to the standard input.

`>&p`
: The output to the coprocess is moved to the standard output.

If one of the above is preceded by a digit, then the file descriptor referred to is that specified by the digit (instead of the default 0 or 1). The order in which redirections are specified is significant. The shell evaluates each redirection in terms of the (*file descriptor*, *file*) association at the time of evaluation. For example:

. . . `1>`*fname* `2>&1`

first associates file descriptor 1 with file *fname*. It then associates file descriptor 2 with the file associated with file descriptor 1 (that is, *fname*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and then file descriptor 1 would be associated with file *fname*.

If the user tries to open a file descriptor for writing more than once, the shell opens the file descriptor as a pipe to a process that copies its input to all the specified outputs, similar to `tee(1)`, provided the `MULTIOS` option is set. Thus:

```
date >foo >bar
```

writes the date to two files, named '`foo`' and '`bar`'. Note that a pipe is an implicit indirection; thus

```
date >foo | cat
```

writes the date to the file '`foo`', and also pipes it to `cat`.

If the `MULTIOS` option is set, the word after a redirection operator is also subjected to filename generation (globbing). Thus

```
: > *
```

will truncate all files in the current directory, assuming there's at least one. (Without the `MULTIOS` option, it would create an empty file called '`*`'.)

If the user tries to open a file descriptor for reading more than once, the shell opens the file descriptor as a pipe to a process that copies all the specified inputs to its output in the order specified, similar to `cat(1)`, provided the `MULTIOS` option is set. Thus

```
sort <foo <fubar
```

or even

```
sort <f{oo,ubar}
```

is equivalent to '`cat foo fubar | sort`'. Similarly, you can do

```
echo exit 0 >> *.sh
```

Note that a pipe is an implicit indirection; thus

```
cat bar | sort <foo
```

is equivalent to '`cat bar foo | sort`' (note the order of the inputs).

If the `MULTIOS` option is unset, each redirection replaces the previous redirection for that file descriptor. However, all files redirected to are actually opened, so

```
echo foo > bar > baz
```

when `MULTIOS` is unset will truncate `bar`, and write `foo` into `baz`.

If a simple command consists of one or more redirection operators and zero or more parameter assignments, but no command name, the command `cat` is assumed. Thus

```
< file
```

prints the contents of `file`.

If a command is followed by `&` and job control is not active, then the default standard input for the command is the empty file '`/dev/null`'. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

8 Command Execution

If a command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, the function is invoked as described below (see Chapter 9 [Functions], page 33). If there exists a shell builtin by that name, the builtin is invoked.

Otherwise, the shell searches each element of `path` for a directory containing an executable file by that name. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.

If execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a shell script. `/bin/sh` is spawned to execute it. If the program is a file beginning with `#!`, the remainder of the first line specifies an interpreter for the program. The shell will execute the specified interpreter on operating systems that do not handle this executable format in the kernel.

9 Functions

The `function` reserved word is used to define shell functions. Shell functions are read in and stored internally. Alias names are resolved when the function is read. Functions are executed like commands with the arguments passed as positional parameters.

Functions execute in the same process as the caller and share all files and the present working directory with the caller. A trap on `EXIT` set inside a function is executed after the function completes in the environment of the caller.

The `return` builtin is used to return from function calls.

Function identifiers can be listed with the `functions` builtin. Functions can be undefined with the `unfunction` builtin.

The following functions, if defined, have special meaning to the shell:

`chpwd` Executed whenever the current working directory is changed.

`precmd` Executed before each prompt.

`periodic` If the parameter `PERIOD` is set, this function is executed every `PERIOD` seconds, just before a prompt.

`TRAP`*xxx* If defined and non-null, this function will be executed whenever the shell catches a signal `SIG`*xxx*, where *xxx* is a signal name as specified for the `kill` builtin (see Chapter 17 [Shell Builtin Commands], page 75). In addition, `TRAPZERR` is executed whenever a command has a non-zero exit status, `TRAPDEBUG` is executed after each command, and `TRAPEXIT` is executed when the shell exits, or when the current function exits if defined inside a function. If a function of this form is defined and null, the shell and processes spawned by it will ignore `SIG`*xxx*.

10 Jobs & Signals

If the `MONITOR` option is set, an interactive shell associates a *job* with each pipeline. It keeps a table of current jobs, printed by the `jobs` command, and assigns them small integer numbers. When a job is started asynchronously with `&`, the shell prints a line which looks like:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If a job is started with `&|` or `&!`, then that job is immediately disowned. After startup, it does not have a place in the job table, and is not subject to the job control features described here.

If you are running a job and wish to do something else you may hit the key *`^Z`* (control-Z) which sends a `TSTP` signal to the current job. The shell will then normally indicate that the job has been *suspended*, and print another prompt. You can then manipulate the state of this job, putting it into the background with the `bg` command, or run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`. A *`^Z`* takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will suspend if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command `stty tostop`. If you set this tty option, then background jobs will suspend when they try to produce output, like they do when they try to read input.

There are several ways to refer to jobs in the shell. A job can be referred to by the process id of any process of the job or by one of the following:

`%`*number* The job with the given number.

`%`*string* Any job whose command line begins with *string*.

`%?`*string* Any job whose command line contains *string*.

`%%` Current job.

`%+` Equivalent to `%%`.

`%-` Previous job.

The shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible. If `notify` is not set, it waits until just before it prints a prompt before it informs you.

When the monitor mode is on, each background job that completes triggers any trap set for `CHLD`.

When you try to leave the shell while jobs are running or suspended, you will be warned that '`You have suspended (running) jobs`'. You may use the `jobs` command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time; the suspended jobs will be terminated, and the running jobs will be sent a `SIGHUP` signal. To avoid having the shell terminate the running jobs, either use the `nohup(1)` command or the `disown` builtin (see Chapter 17 [Shell Builtin Commands], page 75).

The `INT` and `QUIT` signals for an invoked command are ignored if the command is followed by `&` and the job `monitor` option is not active. Otherwise, signals have the values inherited by the shell from its parent (but See Chapter 9 [Functions], page 33, for the `TRAP`*xxx* special functions).

11 Arithmetic Evaluation

An ability to perform integer arithmetic is provided with the builtin `let`. Evaluations are performed using *long* arithmetic. A leading `0x` or `0X` denotes hexadecimal. Otherwise, numbers are of the form `[`*base*`#]`*n* where *base* is a decimal number between two and thirty-six representing the arithmetic base and *n* is a number in that base (for example, `16#ff` is 255 in hexadecimal). If *base* is omitted then base 10 is used. For backwards compatibility the form `[16]ff` is also accepted.

An arithmetic expression uses nearly the same syntax, precedence, and associativity of expressions in C. The following operators are supported (listed in decreasing order of precedence):

`+ - ! ~ ++ --`
: Unary plus/minus, logical NOT, complement, {pre,post}{in,de}crement

`<< >>`
: Bitwise shift left, right.

`&`
: Bitwise AND

`^`
: Bitwise XOR

`|`
: Bitwise OR

`**`
: Exponentiation

`* / %`
: Multiplication, division, modulus (remainder)

`+ -`
: Addition, subtraction

`< > <= >=`
: Comparison

`== !=`
: Equality and inequality

`&&`
: Logical AND

`|| ^^`
: Logical OR, XOR

`? :`
: Ternary operator

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

`,`
: Comma operator

The operators `&&`, `||`, `&&=`, and `||=` are short-circuiting, and only one of the latter two expressions in a ternary operator is evaluated. Note the precedence of the bitwise AND, OR, and XOR operators.

An expression of the form `#\x` where `x` is any character gives the ASCII value of this character. An expression of the form `#foo` gives the ASCII value of the first character of the value of the parameter `foo`.

Named parameters and subscripted arrays can be referenced by name within an arithmetic expression without using the parameter substitution syntax.

An internal integer representation of a named parameter can be specified with the `integer` builtin. Arithmetic evaluation is performed on the value of each assignment to a named parameter declared integer in this manner.

Since many of the arithmetic operators require quoting, an alternative form of the `let` command is provided. For any command which begins with a `((`, all the characters until a matching `))` are treated as a quoted expression. More precisely, `((` ... `))` is equivalent to `let "..."`.

12 Conditional Expressions

A *conditional expression* is used with the `[[` compound command to test attributes of files and to compare strings. Each expression can be constructed from one or more of the following unary or binary expressions:

`-a` *file* True if *file* exists.

`-b` *file* True if *file* exists and is a block special file.

`-c` *file* True if *file* exists and is a character special file.

`-d` *file* True if *file* exists and is a directory.

`-e` *file* True if *file* exists.

`-f` *file* True if *file* exists and is an ordinary file.

`-g` *file* True if *file* exists and has its setgid bit set.

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

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

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

`-o` *option* True if option named *option* is on. *option* may be a single character, in which case it is a single letter option name. See Section 16.1 [Specifying Options], page 65.

`-p` *file* True if *file* exists and is a FIFO special file or a pipe.

`-r` *file* True if *file* exists and is readable by the current process.

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

`-t` *fd* True if file descriptor number *fd* is open and associated with a terminal device (note: *fd* is not optional).

`-u` *file* True if *file* exists and has its setuid bit set.

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

`-x` *file* True if *file* exists and is executable by current process. If *file* exists and is a directory, then the current process has permission to search in the directory.

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

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

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

`-G` *file* True if *file* exists and its group matches the effective group id of this process.

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

`-N` *file* True if *file* exists and its access time is not newer than its modification time.

file1 `-nt` *file2*
True if *file1* exists and is newer than *file2*.

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

file1 `-ef` *file2*
True if *file1* and *file2* exist and refer to the same file.

string `==` *pattern*
string `=` *pattern*
: True if *string* matches *pattern*. The first form is the preferred one. The other form is for backward compatibility and should be considered obsolete.

string `!=` *pattern*
: True if *string* does not match *pattern*.

string1 `<` *string2*
: True if *string1* comes before *string2* based on ASCII value of their characters.

string1 `>` *string2*
: True if *string1* comes after *string2* based on ASCII value of their characters.

exp1 `-eq` *exp2*
: True if *exp1* is equal to *exp2*.

exp1 `-ne` *exp2*
: True if *exp1* is not equal to *exp2*.

exp1 `-lt` *exp2*
: True if *exp1* is less than *exp2*.

exp1 `-gt` *exp2*
: True if *exp1* is greater than *exp2*.

exp1 `-le` *exp2*
: True if *exp1* is less than or equal to *exp2*.

exp1 `-ge` *exp2*
: True if *exp1* is greater than or equal to *exp2*.

`(` *exp* `)`
: True if *exp* is true.

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

exp1 `&&` *exp2*
: True if *exp1* and *exp2* are both true.

exp1 `||` *exp2*
: True if either *exp1* or *exp2* is true.

In each of the above expressions, if *file* is of the form '`/dev/fd/`*n*', where *n* is an integer, then the test is applied to the open file whose descriptor number is *n*, even if the underlying system does not support the '`/dev/fd`' directory.

13 Compatibility

zsh tries to emulate sh or ksh when it is invoked as sh or ksh respectively. In this mode the following parameters are not special: ARGC, argv, cdpath, fignore, fpath, HISTCHARS, mailpath, MANPATH, manpath, path, prompt, PROMPT, PROMPT2, PROMPT3, PROMPT4, psvar, status, watch.

The usual zsh startup/shutdown scripts are not executed. Login shells source '/etc/profile' followed by '$HOME/.profile'. If the ENV environment variable is set on invocation, $ENV is sourced after the profile scripts. The value of ENV is subjected to *parameter expansion*, *command substitution*, and *arithmetic expansion* before being interpreted as a pathname. Note that the PRIVILEGED option also affects the execution of startup files. See Chapter 16 [Options], page 65, for more details.

The following options are set if the shell is invoked as sh or ksh: NO_BAD_PATTERN, NO_BANG_HIST, NO_BG_NICE, NO_EQUALS, NO_FUNCTION_ARGZERO, GLOB_SUBST, NO_HUP, INTERACTIVE_COMMENTS, KSH_ARRAYS, NO_MULTIOS, NO_NOMATCH, RM_STAR_SILENT, POSIX_BUILTINS, SH_FILE_EXPANSION, SH_GLOB, SH_OPTION_LETTERS, SH_WORD_SPLIT. Additionally the KSH_OPTION_PRINT, LOCAL_OPTIONS, PROMPT_SUBST and SINGLE_LINE_ZLE options are set if zsh is invoked as ksh and the IGNORE_BRACES and BSD_ECHO options are set if zsh is invoked as sh.

14 Zsh Line Editor

If the `ZLE` option is set (it is by default) and the shell input is attached to the terminal, the user is allowed to edit command lines.

There are two display modes. The first, multi-line mode, is the default. It only works if the `TERM` parameter is set to a valid terminal type that can move the cursor up. The second, single line mode, is used if `TERM` is invalid or incapable of moving the cursor up, or if the `SINGLE_LINE_ZLE` option is set. This mode is similar to ksh, and uses no termcap sequences. If `TERM` is '`emacs`', the `ZLE` option will be unset by the shell.

14.1 Bindings

Command bindings may be set using the `bindkey` builtin. There are two keymaps; the main keymap and the alternate keymap. The alternate keymap is bound to vi command mode. The main keymap is bound to emacs mode by default. To bind the main keymap to vi insert mode, use `bindkey -v`. However, if either of the `VISUAL` or `EDITOR` environment variables contains the string '`vi`' when the shell starts up the main keymap will be bound to vi insert mode by default.

The following is a list of all the key commands and their default bindings in emacs mode, vi command mode and vi insert mode.

14.2 Movement

`vi-backward-blank-word (unbound) (`*`B`*`) (unbound)`
: Move backward one word, where a word is defined as a series of non-blank characters.

`backward-char (`*`^B ESC-[D`*`) (unbound)`
: Move backward one character.

`vi-backward-char (unbound) (`*`^H h ^?`*`) (unbound)`
: Move backward one character, without changing lines.

`backward-word (`*`ESC-B ESC-b`*`) (unbound) (unbound)`
: Move to the beginning of the previous word.

`emacs-backward-word`
: Move to the beginning of the previous word.

`vi-backward-word (unbound) (`*`b`*`) (unbound)`
: Move to the beginning of the previous word, vi-style.

`beginning-of-line (`*`^A`*`) (unbound) (unbound)`
: Move to the beginning of the line. If already at the beginning of the line, move to the beginning of the previous line, if any.

`vi-beginning-of-line`
: Move to the beginning of the line, without changing lines.

`end-of-line (`*`^E`*`) (unbound) (unbound)`
: Move to the end of the line. If already at the end of the line, move to the end of the next line, if any.

`vi-end-of-line` (unbound) (*$*) (unbound)
: Move to the end of the line. If an argument is given to this command, the cursor will be moved to the end of the line (argument - 1) lines down.

`vi-forward-blank-word` (unbound) (*W*) (unbound)
: Move forward one word, where a word is defined as a series of non-blank characters.

`vi-forward-blank-word-end` (unbound) (*E*) (unbound)
: Move to the end of the current word, or, if at the end of the current word, to the end of the next word, where a word is defined as a series of non-blank characters.

`forward-char` (*^F ESC-[C*) (unbound) (unbound)
: Move forward one character.

`vi-forward-char` (unbound) (⟨SPACE⟩ *l*) (unbound)
: Move forward one character.

`vi-find-next-char` (*^X^F*) (*f*) (unbound)
: Read a character from the keyboard, and move to the next occurrence of it in the line.

`vi-find-next-char-skip` (unbound) (*t*) (unbound)
: Read a character from the keyboard, and move to the position just before the next occurrence of it in the line.

`vi-find-prev-char` (unbound) (*F*) (unbound)
: Read a character from the keyboard, and move to the previous occurrence of it in the line.

`vi-find-prev-char-skip` (unbound) (*T*) (unbound)
: Read a character from the keyboard, and move to the position just after the previous occurrence of it in the line.

`vi-first-non-blank` (unbound) (*^*) (unbound)
: Move to the first non-blank character in the line.

`vi-forward-word` (unbound) (*w*) (unbound)
: Move forward one word, vi-style.

`forward-word` (*ESC-F ESC-f*) (unbound) (unbound)
: Move to the beginning of the next word. The editor's idea of a word is specified with the `WORDCHARS` parameter.

`emacs-forward-word`
: Move to the end of the next word.

`vi-forward-word-end` (unbound) (*e*) (unbound)
: Move to the end of the next word.

`vi-goto-column` (*ESC-|*) (*|*) (unbound)
: Move to the column specified by the numeric argument.

`vi-goto-mark` (unbound) (*‘*) (unbound)
: Move to the specified mark.

`vi-goto-mark-line` (unbound) (*’*) (unbound)
: Move to the beginning of the line containing the specified mark.

`vi-repeat-find` (unbound) (*;*) (unbound)
: Repeat the last `vi-find` command.

`vi-rev-repeat-find` (unbound) (*,*) (unbound)
: Repeat the last `vi-find` command in the opposite direction.

14.3 History Control

`beginning-of-buffer-or-history` (*ESC-*<) (unbound) (unbound)
: Move to the beginning of the buffer, or if already there, move to the first event in the history list.

`beginning-of-line-hist`
: Move to the beginning of the line. If already at the beginning of the buffer, move to the previous history line.

`beginning-of-history`
: Move to the first event in the history list.

`down-line-or-history` (*^N ESC-[B*) (*j*) (unbound)
: Move down a line in the buffer, or if already at the bottom line, move to the next event in the history list.

`vi-down-line-or-history` (unbound) (+) (unbound)
: Move down a line in the buffer, or if already at the bottom line, move to the next event in the history list. Then move to the first non-blank character on the line.

`down-line-or-search`
: Move down a line in the buffer, or if already at the bottom line, search forward in the history for a line beginning with the first word in the buffer.

`down-history` (unbound) (*^N*) (unbound)
: Move to the next event in the history list.

`history-beginning-search-backward`
: Search backward in the history for a line beginning with the current line up to the cursor. This leaves the cursor in its original position.

`end-of-buffer-or-history` (*ESC-*>) (unbound) (unbound)
: Move to the end of the buffer, or if already there, move to the last event in the history list.

`end-of-line-hist`
: Move to the end of the line. If already at the end of the buffer, move to the next history line.

`end-of-history`
: Move to the last event in the history list.

`vi-fetch-history` (unbound) (*G*) (unbound)
: Fetch the history line specified by the numeric argument. This defaults to the current history line (i.e. the one that isn't history yet).

`history-incremental-search-backward` (*^R ^Xr*) (unbound) (unbound)
: Search backward incrementally for a specified string. The string may begin with `^` to anchor the search to the beginning of the line. A restricted set of editing functions is available in the mini-buffer. An interrupt signal, as defined by the stty setting, will stop the search and go back to the original line. An undefined key will have the same effect. The supported functions are: `backward-delete-char`, `vi-backward-delete-character`, `clearscreen`, `redisplay`, `quoted-insert`, `vi-quoted-insert`, `accept-and-hold`, `accept-and-infer-next-history`, `accept-line` and `accept-line-and-down-history`; `magic-space` just inserts a space. `vi-cmd-mode` toggles between the main and alternate key bindings; the main key bindings (insert mode) will be selected initially. Any string that is bound to an out-string (via `bindkey -s`) will behave as if out-string were typed directly. Typing the binding of `history-incremental-search-backward` will get the next occurrence of the contents of the mini-buffer. Typing the

binding of `history-incremental-search-forward` inverts the sense of the search. The direction of the search is indicated in the mini-buffer. Any single character that is not bound to one of the above functions, or `self-insert` or `self-insert-unmeta` will have the same effect but the function will be executed.

`history-incremental-search-forward` (*^S ^Xs*) (unbound) (unbound)
: Search forward incrementally for a specified string. The string may begin with ^ to anchor the search to the beginning of the line. The functions available in the mini-buffer are the same as for `history-incremental-search-backward`.

`history-search-backward` (*ESC-P ESC-p*) (unbound) (unbound)
: Search backward in the history for a line beginning with the first word in the buffer.

`vi-history-search-backward` (unbound) (*/*) (unbound)
: Search backward in the history for a specified string. The string may begin with ^ to anchor the search to the beginning of the line. A restricted set of editing functions is available in the mini-buffer. An interrupt signal, as defined by the stty setting, will stop the search. The functions available in the mini-buffer are: `accept-line`, `vi-cmd-mode` (treated the same as `acceptline`), `backward-delete-char`, `vi-backward-delete-char`, `backward-kill-word`, `vi-backward-kill-word`, `clear-screen`, `redisplay`, `magic-space` (treated as a space), `quoted-insert` and `vi-quoted-insert`. Any string that is not bound to an out-string (via `bindkey -s`) will behave as if out-string were typed directly. Any other character that is not bound to `self-insert` or `self-insert-unmeta` will beep and be ignored. If the function is called from vi command mode, the bindings of the current insert mode will be used.

`history-search-forward` (*ESC-N ESC-n*) (unbound) (unbound)
: Search forward in the history for a line beginning with the first word in the buffer.

`vi-history-search-forward` (unbound) (*?*) (unbound)
: Search forward in the history for a specified string. The string may begin with ^ to anchor the search to the beginning of the line. The functions available in the mini-buffer are the same as for `vi-history-search-backward`.

`infer-next-history` (*^X^N*) (unbound) (unbound)
: Search in the history for a line matching the current one and fetch the event following it.

`insert-last-word` (*ESC-_ ESC-.*) (unbound) (unbound)
: Insert the last word from the previous history event at the cursor position. If a positive numeric argument is given, insert that word from the end of the previous history event. If the argument is zero or negative insert that word from the left (zero inserts the previous command word).

`vi-repeat-search` (unbound) (*n*) (unbound)
: Repeat the last vi history search.

`vi-rev-repeat-search` (unbound) (*N*) (unbound)
: Repeat the last vi history search, but in reverse.

`up-line-or-history` (*^P ESC-[A*) (*k*) (unbound)
: Move up a line in the buffer, or if already at the top line, move to the previous event in the history list.

`up-line-or-search`
: Move up a line in the buffer, or if already at the top line, search backward in the history for a line beginning with the first word in the buffer.

`up-history` (unbound) (*^P*) (unbound)
: Move to the previous event in the history list.

`history-beginning-search-forward`
Search forward in the history for a line beginning with the current line up to the cursor. This leaves the cursor at its original position.

14.4 Modifying Text

`vi-add-eol` (unbound) (*A*) (unbound)
Move to the end of the line and enter insert mode.

`vi-add-next` (unbound) (*a*) (unbound)
Enter insert mode after the current cursor position, without changing lines.

`backward-delete-char` (*^H ^?*) (unbound) (unbound)
Delete the character behind the cursor.

`vi-backward-delete-char` (unbound) (*X*) (*^H*)
Delete the character behind the cursor, without changing lines. If in insert mode this won't delete past the point where insert mode was last entered.

`backward-delete-word`
Delete the word behind the cursor.

`backward-kill-line`
Kill from the beginning of the line to the cursor position.

`backward-kill-word` (*^W ESC-^H ESC-^?*) (unbound) (unbound)
Kill the word behind the cursor.

`vi-backward-kill-word` (unbound) (unbound) (*^W*)
Kill the word behind the cursor, without going past the point where insert mode was last entered.

`capitalize-word` (*ESC-C ESC-c*) (unbound) (unbound)
Capitalize the current word and move past it.

`vi-change` (unbound) (*c*) (unbound)
Read a movement command from the keyboard, and kill from the cursor position to the endpoint of the movement. Then enter insert mode. If the command is `vi-change`, kill the current line.

`vi-change-eol` (unbound) (*C*) (unbound)
Kill to the end of the line and enter insert mode.

`vi-change-whole-line` (unbound) (*S*) (unbound)
Kill the current line and enter insert mode.

`copy-region-as-kill` (*ESC-W ESC-w*) (unbound) (unbound)
Copy the area from the cursor to the mark to the kill buffer.

`copy-prev-word` (*ESC-^_*) (unbound) (unbound)
Duplicate the word behind the cursor.

`vi-delete` (unbound) (*d*) (unbound)
Read a movement command from the keyboard, and kill from the cursor position to the endpoint of the movement. If the command is `vi-delete`, kill the current line.

`delete-char`
Delete the character under the cursor.

`vi-delete-char` (unbound) (*x*) (unbound)
Delete the character under the cursor, without going past the end of the line.

`delete-word`
: Delete the current word.

`down-case-word` (*ESC-L ESC-l*) (unbound) (unbound)
: Convert the current word to all lowercase and move past it.

`kill-word` (*ESC-D ESC-d*) (unbound) (unbound)
: Kill the current word.

`gosmacs-transpose-chars`
: Exchange the two characters behind the cursor.

`vi-indent` (unbound) (>) (unbound)
: Indent a number of lines.

`vi-insert` (unbound) (*i*) (unbound)
: Enter insert mode.

`vi-insert-bol` (unbound) (*I*) (unbound)
: Move to the beginning of the line and enter insert mode.

`vi-join` (*^X^J*) (*J*) (unbound)
: Join the current line with the next one.

`kill-line` (*^K*) (unbound) (unbound)
: Kill from the cursor to the end of the line.

`vi-kill-line` (unbound) (unbound) (*^U*)
: Kill from the cursor back to wherever insert mode was last entered.

`vi-kill-eol` (unbound) (*D*) (unbound)
: Kill from the cursor to the end of the line.

`kill-region`
: Kill from the cursor to the mark.

`kill-buffer` (*^X^K*) (unbound) (unbound)
: Kill the entire buffer.

`kill-whole-line` (*^U*) (unbound) (unbound)
: Kill the current line.

`vi-match-bracket` (*^X^B*) (*%*) (unbound)
: Move to the bracket character (one of `{}`, `()`, or `[]`) that matches the one under the cursor. If the cursor is not on a bracket character, move forward without going past the end of the line to find one, and then go to the matching bracket.

`vi-open-line-above` (unbound) (*O*) (unbound)
: Open a line above the cursor and enter insert mode.

`vi-open-line-below` (unbound) (*o*) (unbound)
: Open a line below the cursor and enter insert mode.

`vi-oper-swap-case`
: Read a movement command from the keyboard, and swap the case of all characters from the cursor position to the endpoint of the movement. If the movement command is `vi-oper-swap-case`, swap the case of all characters on the current line.

`overwrite-mode` (*^X^O*) (unbound) (unbound)
: Toggle between overwrite mode and insert mode.

`vi-put-before` (unbound) (*P*) (unbound)
: Insert the contents of the kill buffer before the cursor. If the kill buffer contains a sequence of lines (as opposed to characters), paste it above the current line.

`vi-put-after` (unbound) (*p*) (unbound)
: Insert the contents of the kill buffer after the cursor. If the kill buffer contains a sequence of lines (as opposed to characters), paste it below the current line.

`quoted-insert` (*^V*) (unbound) (unbound)
: Insert the next character typed into the buffer literally. An interrupt character will not be inserted.

`vi-quoted-insert` (unbound) (unbound) (*^Q ^V*)
: Display a ^ at the current position, and insert the next character typed into the buffer literally. An interrupt character will not be inserted.

`quote-line` (*ESC-'*) (unbound) (unbound)
: Quote the current line; that is, put a `'` character at the beginning and the end, and convert all `'` characters to `\'`.

`quote-region` (*ESC-"*) (unbound) (unbound)
: Quote the region from the cursor to the mark.

`vi-replace` (unbound) (*R*) (unbound)
: Enter overwrite mode.

`vi-repeat-change` (unbound) (*.*) (unbound)
: Repeat the last vi mode text modification. If a count was used with the modification, it is remembered. If a count is given to this command, it overrides the remembered count, and is remembered for future uses of this command. The cut buffer specification is similarly remembered.

`vi-replace-chars` (unbound) (*r*) (unbound)
: Replace the character under the cursor with a character read from the keyboard.

`self-insert` (printable characters) (unbound) (printable characters and some control characters)
: Put a character in the buffer at the cursor position.

`self-insert-unmeta` (*ESC-^I ESC-^J ESC-^M*) (unbound) (unbound)
: Put a character in the buffer after stripping the meta bit and converting `^M` to `^J`.

`vi-substitute` (unbound) (*s*) (unbound)
: Substitute the next character(s).

`vi-swap-case` (unbound) (*~*) (unbound)
: Swap the case of the character under the cursor and move past it.

`transpose-chars` (*^T*) (unbound) (unbound)
: Exchange the two characters to the left of the cursor if at end of line, else exchange the character under the cursor with the character to the left.

`transpose-words` (*ESC-T ESC-t*) (unbound) (unbound)
: Exchange the current word with the one before it.

`vi-unindent` (unbound) (*<*) (unbound)
: Unindent a number of lines.

`up-case-word` (*ESC-U ESC-u*) (unbound) (unbound)
: Convert the current word to all caps and move past it.

`yank` (*^Y*) (unbound) (unbound)
: Insert the contents of the kill buffer at the cursor position.

`yank-pop` (*ESC-y*) (unbound) (unbound)
: Remove the text just yanked, rotate the kill-ring, and yank the new top. Only works following `yank` or `yank-pop`.

`vi-yank` (unbound) (*y*) (unbound)
: Read a movement command from the keyboard, and copy the region from the cursor position to the endpoint of the movement into the kill buffer. If the command is `vi-yank`, copy the current line.

`vi-yank-whole-line` (unbound) (*Y*) (unbound)
: Copy the current line into the kill buffer.

`vi-yank-eol`
: Copy the region from the cursor position to the end of the line into the kill buffer. Arguably, this is what *Y* should do in vi, but it isn't what it actually does.

14.5 Arguments

`digit-argument` (*ESC-0*. . .*ESC-9*) (1-9) (unbound)
: Start a new numeric argument, or add to the current one. See also `vi-digit-or-beginning-of-line`.

`neg-argument` (*ESC--*) (unbound) (unbound)
: Changes the sign of the following argument.

`universal-argument`
: Multiply the argument of the next command by 4.

14.6 Completion

`accept-and-menu-complete`
: In a menu completion, insert the current completion into the buffer, and advance to the next possible completion.

`complete-word`
: Attempt completion on the current word.

`delete-char-or-list` (*^D*) (unbound) (unbound)
: Delete the character under the cursor. If the cursor is at the end of the line, list possible completions for the current word.

`expand-cmd-path`
: Expand the current command to its full pathname.

`expand-or-complete` (⟨TAB⟩) (unbound) (⟨TAB⟩)
: Attempt shell expansion on the current word. If that fails, attempt completion.

`expand-or-complete-prefix`
: Attempt shell expansion on the current word up to cursor.

`expand-history` (*ESC-SPACE ESC-!*) (unbound) (unbound)
: Perform history expansion on the edit buffer.

`expand-word` (*^X**) (unbound) (unbound)
: Attempt shell expansion on the current word.

`list-choices` (*ESC-^D*) (*^D=*) (*^D*)
: List possible completions for the current word.

`list-expand` (*^Xg ^XG*) (*^G*) (*^G*)
: List the expansion of the current word.

`magic-space`
: Perform history expansion and insert a space into the buffer. This is intended to be bound to ⟨SPACE⟩.

`menu-complete`
: Like `complete-word`, except that menu completion is used. See Chapter 16 [Options], page 65, for the `MENU_COMPLETE` option.

`menu-expand-or-complete`
: Like `expand-or-complete`, except that menu completion is used.

`reverse-menu-complete`
: See Chapter 16 [Options], page 65, for the `MENU_COMPLETE` option.

14.7 Miscellaneous

`accept-and-hold (ESC-A ESC-a) (unbound) (unbound)`
: Push the contents of the buffer on the buffer stack and execute it.

`accept-and-infer-next-history`
: Execute the contents of the buffer. Then search the history list for a line matching the current one and push the event following onto the buffer stack.

`accept-line (^J ^M) (^J ^M) (^J ^M)`
: Execute the contents of the buffer.

`accept-line-and-down-history (^O) (unbound) (unbound)`
: Execute the current line, and push the next history event on the the buffer stack.

`vi-cmd-mode (^X^V) (unbound) (^[)`
: Enter command mode; that is, use the alternate keymap. Yes, this is bound by default in emacs mode.

`vi-caps-lock-panic`
: Hang until any lowercase key is pressed. This is for vi users without the mental capacity to keep track of their caps lock key (like the author).

`clear-screen (^L ESC-^L) (^L) (^L)`
: Clear the screen and redraw the prompt.

`describe-key-briefly`
: Waits for keypress, then prints the function bound to the pressed key.

`exchange-point-and-mark (^X^X) (unbound) (unbound)`
: Exchange the cursor position with the position of the mark.

`execute-named-cmd (ESC-x) (unbound) (unbound)`
: Read the name of a editor command and execute it. A restricted set of editing functions is available in the mini-buffer. An interrupt signal, as defined by the stty setting, will abort the function. The allowed functions are: `backward-delete-char`, `vi-backward-delete-char`, `clear-screen`, `redisplay`, `quoted-insert`, `vi-quoted-insert`, `kill-region` (kills the last word), `backward-kill-word`, `vi-backward-kill-word`, `kill-whole-line`, `vi-kill-line`, `backward-kill-line`, `list-choices`, `delete-char-or-list`, `complete-word`, `expand-or-complete`, `expand-or-complete-prefix`, `accept-line`, and `vi-cmd-mode` (treated the same as accept line). The ⟨SPC⟩ and ⟨TAB⟩ characters, if not bound to one of these functions, will complete the name and then list the possibilities if the `AUTO_LIST` option is set. Any string that is bound to an out-string (via `bindkey -s`) will behave as if out-string were typed directly. Any other character

that is not bound to `self-insert` or `self-insert-unmeta` will beep and be ignored. If the function is called from vi command mode, the bindings of the current insert mode will be used.

`execute-last-named-cmd` (*ESC-z*) (unbound) (unbound)
: Redo the last function executed with `execute-named-cmd`.

`get-line` (*ESC-G ESC-g*) (unbound) (unbound)
: Pop the top line off the buffer stack and insert it at the cursor position.

`pound-insert` (unbound) (#) (unbound)
: If there is no `#` character at the beginning of the buffer, add one to the beginning of each line. If there is one, remove a `#` from each line that has one. In either case, accept the current line. The `INTERACTIVE_COMMENTS` option must be set for this to have any usefulness.

`vi-pound-insert`
: If there is no `#` character at the beginning of the current line, add one. If there is one, remove it. The `INTERACTIVE_COMMENTS` option must be set for this to have any usefulness.

`push-input`
: Push the entire current multi-line construct onto the buffer stack and return to the top-level (`PS1`) prompt. If the current parser construct is only a single line, this is exactly like `push-line`. Next time the editor starts up or is popped with `get-line`, the construct will be popped off the top of the buffer stack and loaded into the editing buffer.

`push-line` (*^Q ESC-Q ESC-q*) (unbound) (unbound)
: Push the current buffer onto the buffer stack and clear the buffer. Next time the editor starts up, the buffer will be popped off the top of the buffer stack and loaded into the editing buffer.

`push-line-or-edit`
: At the top-level (`PS1`) prompt, equivalent to `push-line`. At a secondary (`PS2`) prompt, move the entire current multi-line construct into the editor buffer. The latter is equivalent to `push-line` followed by `get-line`.

`redisplay` (unbound) (*^R*) (*^R*)
: Redisplays the edit buffer.

`send-break` (*^G ESC-^G*) (unbound) (unbound)
: Abort the current editor function, e.g. `execute-named-command`, or the editor itself, e.g. if you are in `vared`. Otherwise abort the parsing of the current line.

`run-help` (*ESC-H ESC-h*) (unbound) (unbound)
: Push the buffer onto the buffer stack, and execute the command `run-help` *cmd*, where *cmd* is the current command. `run-help` is normally aliased to `man`.

`vi-set-buffer` (unbound) (") (unbound)
: Specify a buffer to be used in the following command. There are 35 buffers that can be specified: the 26 *named* buffers `"a` to `"z` and the nine *queued* buffers `"1` to `"9`. The named buffers can also be specified as `"A` to `"Z`. When a buffer is specified for a cut command, the text being cut replaces the previous contents of the specified buffer. If a named buffer is specified using a capital, the newly cut text is appended to the buffer instead of overwriting it. If no buffer is specified for a cut command, `"1` is used, and the contents of `"1` to `"8` are each shifted along one buffer; the contents of `"9` is lost.

`vi-set-mark` (unbound) (*m*) (unbound)
: Set the specified mark at the cursor position.

`set-mark-command (^@) (unbound) (unbound)`
: Set the mark at the cursor position.

`spell-word (`*ESC-$ ESC-S ESC-s*`) (unbound) (unbound)`
: Attempt spelling correction on the current word.

`undefined-key (lots o' keys) (lots o' keys) (unbound)`
: Beep.

`undo (`*^_ ^Xu ^X^U*`) (unbound) (unbound)`
: Incrementally undo the last text modification.

`vi-undo-change (unbound) (`*u*`) (unbound)`
: Undo the last text modification. If repeated, redo the modification.

`where-is`
: Read the name of an editor command and and print the listing of key sequences that invoke the specified command.

`which-command (`*ESC-?*`) (unbound) (unbound)`
: Push the buffer onto the buffer stack, and execute the command `which-command` *cmd*, where *cmd* is the current command. `which-command` is normally aliased to `whence`.

`vi-digit-or-beginning-of-line(unbound) (`*0*`) (unbound)`
: If the last command executed was a digit as part of an argument, continue the argument. Otherwise, execute `vi-beginning-of-line`.

15 Parameters

A parameter has a name, a value, and a number of attributes. A name may be any sequence of alphanumeric characters and `_`'s, or the single characters `*`, `@`, `#`, `?`, `-`, `$`, or `!`. The value may be either a scalar (a string), an integer, or an array. To assign a scalar or integer value to a parameter, use the `typeset` builtin. To assign an array value, use '`set -A` *name value* ...'. The value of a parameter may also be assigned by writing:

name`=`*value* ...

If the integer attribute, '`-i`', is set for *name*, the *value* is subject to arithmetic evaluation.

15.1 Array Parameters

The value of an array parameter may be assigned by writing:

name=(*value* ...) ...

Individual elements of an array may be selected using a subscript. A subscript of the form `[`*exp*`]` selects the single element *exp*, where *exp* is an arithmetic expression which will be subject to arithmetic expansion as if it were surrounded by `$((...))`. The elements are numbered beginning with 1 unless the `KSH_ARRAYS` option is set when they are numbered from zero.

A subscript of the form `[*]` or `[@]` evaluates to all elements of an array; there is no difference between the two except when they appear within double quotes. `"$foo[*]"` evaluates to `"$foo[1] $foo[2]` ...", while `"$foo[@]"` evaluates to `"$foo[1]" "$foo[2]"`, etc.

A subscript of the form `[`*exp1*`,`*exp2*`]` selects all elements in the range *exp1* to *exp2*, inclusive. If one of the subscripts evaluates to a negative number, say `-`*n*, then the *n*'th element from the end of the array is used. Thus `$foo[-3]` is the third element from the end of the array `foo`, and `$foo[1,-1]` is the same as `$foo[*]`.

Subscripting may also be performed on non-array values, in which case the subscripts specify a substring to be extracted. For example, if `FOO` is set to `foobar`, then `echo $`*FOO*`[2,5]` prints `ooba`.

Subscripts may be used inside braces used to delimit a parameter name, thus `${foo[2]}` is equivalent to `$foo[2]`. If the `KSH_ARRAYS` option is set, the braced form is the only one that will work, the subscript otherwise not being treated specially.

If a subscript is used on the left side of an assignment the selected range is replaced by the expression on the right side.

If the opening bracket or the comma is directly followed by an opening parenthesis the string up to the matching closing parenthesis is considered to be a list of flags. The flags currently understood are:

`e` This option has no effect and retained for backward compatibility only.

`w` If the parameter subscripted is a scalar, then this flag makes subscription work on a per-word basis instead of characters.

`s:`*string*`:` Defines the *string* that separates words (for use with the `w` flag).

`p` Recognize the same escape sequences as the `print` builtin in the string argument of a subsequent `s` flag.

`f` If the parameter subscripted is a scalar than this flag makes subscription work on a per-line basis instead of characters. This is a shorthand for `pws:\n:`.

`r` If this flag is given the exp is taken as a pattern and the result is the first matching array element, substring or word (if the parameter is an array, if it is a scalar, or if it is a scalar and the `w` flag is given, respectively); note that this is like giving a number: `$foo[(r)??,3]` and `$foo[(r)??,(r)f*]` work.

`R` Like `r`, but gives the last match.

`i` Like `r`, but gives the index of the match instead; this may not be combined with a second argument.

`I` Like `i`, but gives the index of the last match.

`n:`*expr*`:` If combined with `r`, `R`, `i`, or `I`, makes them return the *n*'th or *n*'th last match (if *expr* evaluates to *n*).

15.2 Positional Parameters

Positional parameters are set by the shell on invocation, by the `set` builtin, or by direct assignment. The parameter *n*, where *n* is a number, is the *n*'th positional parameter. The parameters `*`, `@`, and `argv` are arrays containing all the positional parameters; thus `argv[`*n*`]`, is equivalent to simply *n*.

15.3 Parameters Set By The Shell

The following parameters are automatically set by the shell:

`!` The process id of the last background command invoked.

`#` The number of positional parameters in decimal.

`ARGC` Same as `#`. It has no special meaning in sh/ksh compatibility mode.

`$` The process id of this shell.

`-` Flags supplied to the shell on invocation or by the `set` or `setopt` commands.

`*` An array containing the positional parameters.

`argv` Same as `*`. It has no special meaning in sh/ksh compatibility mode.

`@` Same as `argv[@]` but it can be used in sh/ksh compatibility mode.

`?` The exit value returned by the last command.

`status` Same as `?`. It has no special meaning in sh/ksh compatibility mode.

`_` The last argument of the previous command. Also, this parameter is set in the environment of every command executed to the full pathname of the command.

`EGID` The effective group id of the shell process. If you have sufficient privileges, you may change the effective group id of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command with a different effective group id by: `(EGID=egid ; command)`

`EUID` The effective user id of the shell process. If you have sufficient privileges, you may change the effective user id of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command with a different effective user id by: `(EUID=euid ; command)`

`ERRNO` The value of `errno` as set by the most recently failed system call. This value is system dependent and is intended for debugging purposes.

`GID` The group id of the shell process. If you have sufficient privileges, you may change the group id of the shell process by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different group id by: `(GID=gid ; command)`

`HOST` The current hostname.

`LINENO` The line number of the current line within the current script being executed.

`LOGNAME` If the corresponding variable is not set in the environment of the shell, it is initialized to the login name corresponding to the current login session. This parameter is exported by default but this can be disabled using the `typeset` builtin.

`MACHTYPE` The machine type (microprocessor class or machine model), as determined at compile time.

`OLDPWD` The previous working directory.

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

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

`OSTYPE` The operating system, as determined at compile time.

`PPID` The process id of the parent of the shell.

`PWD` The present working directory.

`RANDOM` A random integer from 0 to 32767, newly generated each time this parameter is referenced. The random number generator can be seeded by assigning a numeric value to `RANDOM`.

`SECONDS` The number of seconds since shell invocation. If this parameter is assigned a value, then the value returned upon reference will be the value that was assigned plus the number of seconds since the assignment.

`SHLVL` Incremented by one each time a new shell is started.

`signals` An array containing the names of the signals.

`TTY` The name of the tty associated with the shell, if any.

`TTYIDLE` The idle time of the tty associated with the shell in seconds or -1 if there is no such tty.

`UID` The user id of the shell process. If you have sufficient privileges, you may change the user id of the shell by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different user id by: `(UID=uid ; command)`

`USERNAME` The username corresponding to the user id of the shell process. If you have sufficient privileges, you may change the username (and also the user id and group id) of the shell by assigning to this parameter. Also (assuming sufficient privileges), you may start a single command under a different username (and user id and group id) by: `(USERNAME=username ; command)`

VENDOR
: The vendor, as determined at compile time.

ZSHNAME
ZSH_NAME
: Expands to the basename of the command used to invoke this instance of zsh.

ZSH_VERSION
: The version number of this zsh.

15.4 Parameters Used By The Shell

The following parameters are used by the shell:

ARGV0
: If exported, its value is used as argv[0] of external commands. Usually used in constructs like '`ARGV0=emacs nethack`'.

BAUD
: The baud rate of the current connection. Used by the line editor update mechanism to compensate for a slow terminal by delaying updates until necessary. This may be profitably set to a lower value in some circumstances, e.g. for slow modems dialing into a communications server which is connected to a host via a fast link; in this case, this variable would be set by default to the speed of the fast link, and not the modem. This parameter should be set to the baud rate of the slowest part of the link for best performance. The compensation mechanism can be turned off by setting the variable to zero.

cdpath (CDPATH)
: An array (colon-separated list) of directories specifying the search path for the `cd` command.

COLUMNS
: The number of columns for this terminal session. Used for printing select lists and for the line editor.

DIRSTACKSIZE
: The maximum size of the directory stack. If the stack gets larger than this, it will be truncated automatically. This is useful with the `AUTO_PUSHD` option.

FCEDIT
: The default editor for the `fc` builtin.

fignore (FIGNORE)
: An array (colon-separated list) containing the suffixes of files to be ignored during filename completion. However, if the completion generates only files which would match if this variable would be ignored, than these files are completed anyway.

fpath (FPATH)
: An array (colon-separated list) of directories specifying the search path for function definitions. This path is searched when a function with the '`-u`' attribute is referenced. If an executable file is found, then it is read and executed in the current environment.

histchars
: Three characters used by the shell's history and lexical analysis mechanism. The first character signals the start of a history substitution (default `!`). The second character signals the start of a quick history substitution (default `^`). The third character is the comment character (default `#`).

HISTCHARS
: Depreciated. Use `histchars`.

HISTFILE
: The file to save the history in when an interactive shell exits. If unset, the history is not saved.

`HISTSIZE` The maximum size of the history list.

`HOME` The default argument for the `cd` command.

`IFS` Internal field separators, normally space, tab, and newline, that are used to separate words which result from command or parameter substitution and words read by the `read` builtin. Any characters from the set space, tab and newline that appear in the `IFS` are called *IFS white space*. One or more IFS white space characters or one non-IFS white space character together with any adjacent IFS white space character delimit a field. If an IFS white space character appears twice consecutively in the `IFS`, this character is treated as if it were not an IFS white space character.

`KEYTIMEOUT`
The time the shell waits, in hundredths of seconds, for another key to be pressed when reading bound multi-character sequences.

`LANG` This variable determines the locale category for any category not specifically selected via a variable starting with `LC_`.

`LC_ALL` This variable overrides the value of the `LANG` variable and the value of any of the other variables starting with `LC_`.

`LC_COLLATE`
This variable determines the locale category for character collation information within ranges in glob brackets and for sorting.

`LC_CTYPE` This variable determines the locale category for character handling functions.

`LC_MESSAGES`
This variable determines the language in which messages should be written. Note that zsh does not use message catalogs.

`LC_TIME` This variable determines the locale category for date and time formatting in promt escape sequences.

`LINES` The number of lines for this terminal session. Used for printing select lists and for the line editor.

`LISTMAX` In the line editor, the number of filenames to list without asking first. If set to zero, the shell asks only if the listing would scroll off the screen.

`LOGCHECK` The interval in seconds between checks for login/logout activity using the `watch` parameter.

`MAIL` If this parameter is set and `mailpath` is not set, the shell looks for mail in the specified file.

`MAILCHECK`
The interval in seconds between checks for new mail.

`mailpath (MAILPATH)`
An array (colon-separated list) of filenames to check for new mail. Each filename can be followed by a `?` and a message that will be printed. The message will undergo parameter expansion, command substitution and arithmetic substitution with the variable `$_` defined as the name of the file that has changed. The default message is '`You have new mail`'. If an element is a directory instead of a file the shell will recursively check every file in every subdirectory of the element.

`manpath (MANPATH)`
An array (colon-separated list) whose value is not used by the shell. The `manpath` array can be useful, however, since setting it also sets `MANPATH`, and vice versa.

NULLCMD
The command name to assume if a redirection is specified with no command. Defaults to cat. For sh/ksh-like behaviour, change this to :. For csh-like behaviour, unset this parameter; the shell will print an error message if null commands are entered.

path (PATH)
An array (colon-separated list) of directories to search for commands. When this parameter is set, each directory is scanned and all files found are put in a hash table.

POSTEDIT
This string is output whenever the line editor exits. It usually contains termcap strings to reset the terminal.

PS1
The primary prompt string, printed before a command is read; the default is '%m%# '. If the escape sequence takes an optional integer, it should appear between the % and the next character of the sequence. The following escape sequences are recognized:

%%
A %.

%)
A).

%d
%/
Present working directory ($PWD).

%~
$PWD. If it has a named directory as its prefix, that part is replaced by a ~ followed by the name of the directory. If it starts with $HOME, that part is replaced by a ~.

%c
%.
%C
Trailing component of $PWD. An integer may follow the % to get more than one component. Unless %C is used, tilde expansion is performed first.

!
%h
%!
Current history event number.

%M
The full machine hostname.

%m
The hostname up to the first '.'. An integer may follow the % to specify how many components of the hostname are desired.

%S (%s)
Start (stop) standout mode.

%U (%u)
Start (stop) underline mode.

%B (%b)
Start (stop) boldface mode.

%t
%@
Current time of day, in 12-hour, am/pm format.

%T
Current time of day, in 24-hour format.

%*
Current time of day in 24-hour format, with seconds.

%n
$USERNAME.

%w
The date in day-dd format.

%W
The date in mm/dd/yy format.

%D
The date in yy-mm-dd format.

%D{*string*}
string is formatted using the strftime function. See strftime(3) for more details, if your system has it.

%l
The line (tty) the user is logged in on.

`%?` The return code of the last command executed just before the prompt.

`%_` The status of the parser, i.e. the shell constructs (like `if` and `for`) that have been started on the command line. If given an integer number, that many strings will be printed; zero or no integer means print as many as there are.

`%E` Clears to end of line.

`%#` A `#` if the shell is running as root, a `%` if not. Equivalent to `%(#.#.%%)`

`%v` The value of the first element of the `psvar` array parameter. Following the `%` with an integer gives that element of the array.

`%{...%}` Include a string as a literal escape sequence. The string within the braces should not change the cursor position.

`%(x.`*true-text*`.`*false-text*`)`
: Specifies a ternary expression. The character following the `x` is arbitrary; the same character is used to separate the text for the true result from that for the false result. The separator may not appear in the `true-text`, except as part of a `%` sequence. A `)` may appear in the `false-text` as a `%)`. `true-text` and `false-text` may both contain contain arbitrarily-nested escape sequences, including further ternary expressions. The left parenthesis may be preceded or followed by a positive integer `n`, which defaults to zero. The test character `x` may be any of the following:

  `c`
  `.`
  `~` True if the current path, with prefix replacement, has at least *n* elements.

  `/`
  `C` True if the current absolute path has at least *n* elements.

  `t` True if the time in minutes is equal to *n*.

  `T` True if the time in hours is equal to *n*.

  `d` True if the day of the month is equal to *n*.

  `D` True if the month is equal to *n* (January = 0).

  `w` True if the day of the week is equal to *n* (Sunday = 0).

  `?` True if the exit status of the last command was *n*.

  `#` True if the effective uid of the current process is *n*.

  `g` True if the effective gid of the current process is *n*.

  `L` True if the `SHLVL` parameter is at least *n*.

  `S` True if the `SECONDS` parameter is at least *n*.

  `v` True if the array `psvar` has at least *n* elements.

  `_` True if at least *n* shell constructs were started.

  `%<`*string*`<`
  `%>`*string*`>`
  `%[`*xstring*`]`
  : Specifies truncation behaviour. The third form is equivalent to `%`*xstringx*, i.e. *x* may be `<` or `>`. The numeric argument, which in the third form may appear immediately after the `[`, specifies

the maximum permitted length of the various strings that can be displayed in the prompt. If this integer is zero, or missing, truncation is disabled. Truncation is initially disabled. The forms with `<` truncate at the left of the string, and the forms with `>` truncate at the right of the string. For example, if the current directory is '`/home/pike`', the prompt `%8<..<%/` will expand to '`..e/pike`'. The `string` will be displayed in place of the truncated portion of any string. In this string, the terminating character (`<`, `>` or `]`), or in fact any character, may be quoted by a preceding `\`. `%` sequences are not treated specially. If the `string` is longer than the specified truncation length, it will appear in full, completely replacing the truncated string.

`PS2` The secondary prompt, printed when the shell needs more information to complete a command. Recognizes the same escape sequences as `$PS1`. The default is '`%_> `'.

`PS3` Selection prompt used within a `select` loop. Recognizes the same escape sequences as `PS1`. The default is '`?# `'.

`PS4` The execution trace prompt. Default is '`+ `'.

`PROMPT`
`PROMPT2`
`PROMPT3`
`PROMPT4` Same as `PS1`, `PS2`, `PS3`, and `PS4`, respectively. These parameters do not have any special meaning in sh/ksh compatibility mode.

`psvar (PSVAR)`
An array (colon-separated list) whose first nine values can be used in `PROMPT` strings. Setting `psvar` also sets `PSVAR`, and vice versa.

`prompt` Same as `PS1`. It has no special meaning in sh/ksh compatibility mode.

`READNULLCMD`
The command name to assume if a single input redirection is specified with no command. Defaults to `more`.

`REPORTTIME`
If nonzero, commands whose combined user and system execution times (measured in seconds) are greater than this value have timing statistics printed for them.

`RPROMPT`
`RPS1` This prompt is displayed on the right-hand side of the screen when the primary prompt is being displayed on the left. This does not work if the `SINGLELINEZLE` option is set. Recognizes the same escape sequences as `PROMPT`.

`SAVEHIST` The maximum number of history events to save in the history file.

`SPROMPT` The prompt used for spelling correction. The sequence `%R` expands to the string which presumably needs spelling correction, and `%r` expands to the proposed correction. All other `PROMPT` escapes are also allowed.

`STTY` If this parameter is set in a command's environment, the shell runs the `stty` command with the value of this parameter as arguments in order to set up the terminal before executing the command. The modes apply only to the command, and are reset when it finishes or is suspended. If the command is suspended and continued later with the `fg` or `wait` builtins it will see the modes specified by `STTY`, as if it were not suspended. This (intentionally) does not apply if the command is continued via `kill -CONT`. `STTY` is ignored if the command is run in the background, or if it is in the environment of the shell but not explicitly assigned to in the input line. This avoids running `stty` at every external command by accidentally exporting it. Also note that `STTY` should not be used for window size specifications; these will not be local to the command.

`TIMEFMT` The format of process time reports with the `time` keyword. The default is '`%E real %U user %S system %P %J`'. Recognizes the following escape sequences:

- `%` A `%`.
- `%U` CPU seconds spent in user mode.
- `%S` CPU seconds spent in kernel mode.
- `%E` Elapsed time in seconds.
- `%P` The CPU percentage, computed as (%U+%S)/%E.
- `%J` The name of this job.

A star may be inserted between the percent sign and flags printing time. This cause the time to be printed in `hh:mm:ss.ttt` format (hours and minutes are only printed if they are not zero).

`TMOUT` If this parameter is nonzero, the shell will receive an `ALRM` signal if a command is not entered within the specified number of seconds after issuing a prompt. If there is a trap on `SIGALRM`, it will be executed and a new alarm is scheduled using the value of the `TMOUT` parameter after executing the trap. If no trap is set, and the idle time of the terminal is not less than the value of the `TMOUT` parameter, zsh terminates. Otherwise a new alarm is scheduled to `TMOUT` seconds after the last keypress.

`TMPPREFIX`
A pathname prefix which the shell will use for all temporary files. Note that this should include an initial part for the file name as well as any directory names. The default is '`/tmp/zsh`'.

`watch (WATCH)`
An array (colon-separated list) of login/logout events to report. If it contains the single word '`all`', then all login/logout events are reported. If it contains the single word '`notme`', then all login/logout events are reported except for those originating from `$USERNAME`. An entry in this list may consist of a username, an `@` followed by a remote hostname, and a `%` followed by a line (tty). Any or all of these components may be present in an entry; if a login/logout event matches all of them, it is reported.

`WATCHFMT` The format of login/logout reports if the `watch` parameter is set. Default is '`%n has %a %l from %m`'. Recognizes the following escape sequences:

- `%n` The name of the user that logged in/out.
- `%a` The observed action, i.e. '`logged on`' or '`logged off`'.
- `%l` The line (tty) the user is logged in on.
- `%M` The full hostname of the remote host.
- `%m` The hostname up to the first '`.`'. If only the IP address is available or the utmp field contains the name of an X-windows display, the whole name is printed.

  NOTE: The `%m` and `%M` escapes will work only if there is a host name field in the `utmp` on your machine. Otherwise they are treated as ordinary strings.
- `%S (%s)` Start (stop) standout mode.
- `%U (%u)` Start (stop) underline mode.

`%B` (`%b`) Start (stop) boldface mode.

`%t`
`%@` The time, in 12-hour, am/pm format.

`%T` The time, in 24-hour format.

`%w` The date in day-dd format.

`%W` The date in mm/dd/yy format.

`%D` The date in yy-mm-dd format.

`%(x:`*true-text*`:`*false-text*`)`
Specifies a ternary expression. The character following the `x` is arbitrary; the same character is used to separate the text for the true result from that for the false result. Both the separator and the right parenthesis may be escaped with a backslash. Ternary expressions may be nested.

The test character `x` may be any one of `l`, `n`, `m`, or `M`, which indicate a true result if the corresponding escape sequence would return a non-empty value; or it may be `a`, which indicates a true result if the watched user has logged in, or false if he has logged out. Other characters evaluate to neither true nor false; the entire expression is omitted in this case.

If the result is true, then the *true-text* is formatted according to the result above and printed, and the *false-text* is skipped. If false, the *true-text* is skipped, and the *false-text* is formatted and printed. Either or both of the branches may be empty, but both separators must always be present.

`WORDCHARS`
A list of non-alphanumeric characters considered part of a word by the line editor.

`ZDOTDIR` The directory to search for shell startup files (‘`.zshrc`’, etc), if not `$HOME`.

16 Options

16.1 Specifying Options

Options are primarily referred to by name. These names are case insensitive and underscores are ignored. For example, `allexport` is equivalent to `A__lleXP_ort`.

The sense of an option name may be inverted by preceding it with `no`, so `setopt No_Beep` is equivalent to `unsetopt beep`. This inversion can only be done once, so `nonobeep` is *not* a synonym for `beep`. Similarly, `tify` is *not* a synonym for `nonotify` (the inversion of `notify`).

Some options also have one or more single letter names. There are two sets of single letter options: one used by default, and another when the shell is emulating `sh` or `ksh`. The single letter options can be used on the shell command line, or with the `set`, `setopt` and `unsetopt` builtins, as normal Unix options preceded by `-`.

The sense of the single letter options may be inverted by using `+` instead of `-`. Some of the single letter option names refer to an option being off, in which case the inversion of that name refers to the option being on. For example, `+n` is the short name of `exec`, and `-n` is the short name of its inversion, `noexec`.

16.2 Description of Options

`ALL_EXPORT (-a, ksh: -a)`
: All parameters subsequently defined are automatically exported.

`ALWAYS_LAST_PROMPT`
: If unset, key functions that list completions try to return to the last prompt if given a numeric argument. If set, these functions try to return to the last prompt if given no numeric argument.

`ALWAYS_TO_END`
: If a completion with the cursor in the word was started and it results in only one match, the cursor is placed at the end of the word.

`APPEND_HISTORY`
: If this is set, zsh sessions will append their history list to the history file, rather than overwrite it. Thus, multiple parallel zsh sessions will all have their history lists added to the history file, in the order they are killed. See Chapter 17 [Shell Builtin Commands], page 75, for the `fc` command.

`AUTO_CD (-J)`
: If a command is not in the hash table, and there exists an executable directory by that name, perform the `cd` command to that directory.

`AUTO_LIST (-9)`
: Automatically list choices on an ambiguous completion.

`AUTO_MENU`
: Automatically use menu completion after the second consecutive request for completion, for example by pressing the ⟨TAB⟩ key repeatedly. This option is overridden by `MENU_COMPLETE`.

AUTO_NAME_DIRS
: Any parameter that is set to the absolute name of a directory immediately becomes a name for that directory in the usual form `~param`. If this option is not set, the parameter must be used in that form for it to become a name (a command-line completion is sufficient for this).

AUTO_PARAM_KEYS
: If a parameter name was completed and the next character typed is one of those that have to come directly after the name (like `}`, :, etc.), they are placed there automatically.

AUTO_PARAM_SLASH
: If a parameter is completed whose content is the name of a directory, then add a trailing slash.

AUTO_PUSHD (-N)
: Make `cd` push the old directory onto the directory stack.

AUTO_REMOVE_SLASH
: When the last character resulting from a completion is a slash and the next character typed is a word delimiter, remove the slash.

AUTO_RESUME (-W)
: Treat single word simple commands without redirection as candidates for resumption of an existing job.

BAD_PATTERN (+2)
: If a pattern for filename generation is badly formed, print an error. If this option is unset, the pattern will be left unchanged.

BANG_HIST (+K)
: Perform textual history substitution, treating the character `!` specially.

BEEP (+B)
: Beep.

BG_NICE (-6)
: Run all background jobs at a lower priority. This option is set by default.

BRACE_CCL
: Expand expressions in braces which would not otherwise undergo brace expansion to a lexically ordered list of all the characters. See Section 6.6 [Brace Expansion], page 19.

BSD_ECHO
: Make the echo builtin compatible with the BSD `echo(1)` command. This disables backslashed escape sequences in echo strings unless the '`-e`' option is specified.

CDABLE_VARS (-T)
: If the argument to a `cd` command (or an implied `cd` with the `AUTO_CD` option set) is not a directory, and does not begin with a slash, try to expand the expression as if it were preceded by a `~` (see Section 6.1 [Filename Expansion], page 15).

CHASE_LINKS (-w)
: Resolve symbolic links to their true values.

CLOBBER (+C, ksh: +C)
: Allows `>` redirection to truncate existing files, and `>>` to create files. Otherwise `>!` must be used to truncate a file, and `>>!` to create a file.

COMPLETE_ALIASES
: If set, aliases on the command line are not internally substituted before completion is attempted.

`COMPLETE_IN_WORD`
: If unset, the cursor is moved to the end of the word if completion is started. Otherwise it stays where it is and completion is done from both ends.

`CORRECT (-0)`
: Try to correct the spelling of commands.

`CORRECT_ALL (-O)`
: Try to correct the spelling of all arguments in a line.

`CSH_JUNKIE_HISTORY`
: A history reference without an event specifier will always refer to the previous command.

`CSH_JUNKIE_LOOPS`
: Allow loop bodies to take the form '*list*`; end`' instead of '`do` *list*`; done`'.

`CSH_JUNKIE_QUOTES`
: Complain if a quoted expression runs off the end of a line; prevent quoted expressions from containing un-escaped newlines.

`CSH_NULL_GLOB`
: If a pattern for filename generation has no matches, delete the pattern from the argument list; do not report an error unless all the patterns in a command have no matches. Overrides `NULL_GLOB`.

`EQUALS`
: Perform `=` filename substitution.

`ERR_EXIT (-e, ksh: -e)`
: If a command has a non-zero exit status, execute the `ZERR` trap, if set, and exit. This is disabled while running initialization scripts.

`EXEC (+n, ksh: +n)`
: Do execute commands. Without this option, commands are read and checked for syntax errors, but not executed.

`EXTENDED_GLOB`
: Treat the `#`, `~` and `^` characters as part of patterns for filename generation, etc. (An initial unquoted `~` always produces named directory expansion (see Section 6.1 [Filename Expansion], page 15).)

`EXTENDED_HISTORY`
: Save beginning and ending timestamps to the history file. The format of these timestamps is `:<`*beginning time*`>:<`*ending time*`>:<`*command*`>`.

`FLOW_CONTROL`
: If this option is unset, output flow control via start/stop characters (usually assigned to *^S*/*^Q*) is disabled in the shell's editor.

`FUNCTION_ARGZERO`
: When executing a shell function or sourcing a script, set `$0` temporarily to the name of the function/script.

`GLOB (+F, ksh: +f)`
: Perform filename generation.

`GLOB_ASSIGN`
: If this option is set, filename generation (globbing) is performed on the right hand side of scalar parameter assignments of the form *name*`=`*pattern* (e.g. '`param=*`'). If the result has more than one word the parameter will become an array with those words as arguments. This option is provided for backwards compatibility only: globbing is always performed on the right hand side of array assignments of the form '*name*`=(`

value)' (e.g. '`param=(*)`') and this form is recommended for clarity; with this option set, it is not possible to predict whether the result will be an array or a scalar.

`GLOB_COMPLETE`
: When the current word has a glob pattern, do not insert all the words resulting from the expansion but cycle through them like `MENU_COMPLETE`. If no matches are found, a `*` is added to the end of the word, or inserted at the cursor if `COMPLETE_IN_WORD` is set, and completion is attempted again. Using patterns works not only for files but for all completions, such as options, user names, etc.

`GLOB_DOTS (-4)`
: Do not require a leading `.` in a filename to be matched explicitly.

`GLOB_SUBST`
: Treat any characters resulting from parameter substitution as being eligible for file expansion and filename generation, and any characters resulting from command substitution as being eligible for filename generation.

`HASH_CMDS`
: Place the location of each command in the hash table the first time it is executed. If this option is unset, no path hashing will be done at all.

`HASH_DIRS`
: Whenever a command is executed, hash the directory containing it, as well as all directories that occur earlier in the path. Has no effect if `HASH_CMDS` is unset.

`HASH_LIST_ALL`
: Whenever a command completion is attempted, make sure the entire command path is hashed first. This makes the first completion slower.

`HIST_ALLOW_CLOBBER`
: Add `|` to output redirections in the history. This allows history references to clobber files even when `CLOBBER` is unset.

`HIST_BEEP`
: Beep when an attempt is made to access a history entry which isn't there.

`HIST_IGNORE_DUPS (-h)`
: Do not enter command lines into the history list if they are duplicates of the previous event.

`HIST_IGNORE_SPACE (-g)`
: Do not enter command lines into the history list if they begin with a blank.

`HIST_NO_STORE`
: Remove the `history` (`fc -l`) command from the history when invoked.

`HIST_VERIFY`
: Whenever the user enters a line with history substitution, don't execute the line directly; instead, perform history substitution and reload the line into the editing buffer.

`HUP`
: Send the `HUP` signal to running jobs when the shell exits.

`IGNORE_BRACES (-I)`
: Do not perform brace expansion.

`IGNORE_EOF (-7)`
: Do not exit on end-of-file. Require the use of `exit` or `logout` instead.

`INTERACTIVE (-i, ksh: -i)`
: This is an interactive shell. This option is set upon initialisation if the standard input is a tty and commands are being read from standard input. (See the discussion of

SHIN_STDIN.) This heuristic may be overridden by specifying a state for this option on the command line. The value of this option cannot be changed anywhere other than the command line.

INTERACTIVE_COMMENTS (-k)
: Allow comments even in interactive shells.

KSH_ARRAYS
: Emulate ksh array handling as closely as possible. If this option is set, array elements are numbered from zero, an array parameter without subscript refers to the first element instead of the whole array, and braces are required to delimit a subscript (`${path[2]}` rather than just `$path[2]`).

KSH_OPTION_PRINT
: Alters the way options settings are printed.

LIST_AMBIGUOUS
: If this option is set completions are shown only if the completions don't have an unambiguous prefix or suffix that could be inserted in the command line.

LIST_BEEP
: Beep on an ambiguous completion.

LIST_TYPES (-X)
: When listing files that are possible completions, show the type of each file with a trailing identifying mark.

LOCAL_OPTIONS
: If this option is set at the point of return from a shell function, all the options (including this one) which were in force upon entry to the function are restored. Otherwise, only this option and the XTRACE and PRINT_EXIT_VALUE options are restored. Hence if this is explicitly unset by a shell function the other options in force at the point of return will remain so.

LOGIN (-l, ksh: -l)
: This is a login shell.

LONG_LIST_JOBS (-R)
: List jobs in the long format by default.

MAGIC_EQUAL_SUBST
: All unquoted arguments of the form *identifier=expression* appearing after the command name have file expansion (that is, where *expression* has a leading '~' or '=') performed on *expression* as if it were a parameter assignment. The argument is not otherwise treated specially: in other words, it is subsequently treated as a single word, not as an assignment.

MAIL_WARNING (-U)
: Print a warning message if a mail file has been accessed since the shell last checked.

MARK_DIRS (-8, ksh: -X)
: Append a trailing / to all directory names resulting from filename generation (globbing).

MENU_COMPLETE (-Y)
: On an ambiguous completion, instead of listing possibilities or beeping, insert the first match immediately. Then when completion is requested again, remove the first match and insert the second match, etc. When there are no more matches, go back to the first one again. `reverse-menu-complete` may be used to loop through the list in the other direction. This option overrides AUTO_MENU.

`MONITOR (-m, ksh: -m)`
: Allow job control. Set by default in interactive shells.

`MULTIOS`
: Perform implicit `tee`s or `cat`s when multiple redirections are attempted. See Chapter 7 [Redirection], page 27.

`NOMATCH (-3)`
: If a pattern for filename generation has no matches, print an error, instead of leaving it unchanged in the argument list. This also applies to file expansion of an initial `~` or `=`.

`NOTIFY (-5, ksh: -b)`
: Report the status of background jobs immediately, rather than waiting until just before printing a prompt.

`NULL_GLOB (-G)`
: If a pattern for filename generation has no matches, delete the pattern from the argument list instead of reporting an error. Overrides `NOMATCH`.

`NUMERIC_GLOB_SORT`
: If numeric filenames are matched by a filename generation pattern, sort the filenames numerically rather than lexicographically.

`OVER_STRIKE`
: Start up the line editor in overstrike mode.

`PATH_DIRS (-Q)`
: Perform a path search even on command names with slashes in them. Thus if

  '`/usr/local/bin`'

  is in the user's path, and he types '`X11/xinit`', the command

  '`/usr/local/bin/X11/xinit`'

  will be executed (assuming it exists). This applies to the `.` builtin as well as to command execution. Commands explicitly beginning with '`./`' or '`../`' are not subject to path search.

`POSIX_BUILTINS`
: When this option is set the `command` builtin can be used to execute shell builtin commands. Parameter assignments specified before shell functions and special builtins are kept after the command completes unless the special builtin is prefixed with the `command` builtin. Special builtins are `.`, `:`, `break`, `continue`, `declare`, `eval`, `exit`, `export`, `integer`, `local`, `readonly`, `return`, `set`, `shift`, `source`, `times`, `trap` and `unset`.

`PRINT_EXIT_VALUE (-1)`
: Print the exit value of programs with non-zero exit status.

`PRIVILEGED (-p, ksh: -p)`
: Turn on privileged mode. 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. This option disables sourcing user startup files. If zsh is invoked as sh or ksh with this option set, '`/etc/suid_profile`' is sourced (after '`/etc/profile`' on interactive shells). Sourcing '`~/.profile`' is disabled and the contents of the `ENV` variable is ignored. This option cannot be changed using the '`-m`' option of `setopt` and `unsetopt` and changing it inside a function always changes it globally regardless of the `LOCAL_OPTIONS` option.

`PROMPT_CR (+V)`
: Print a carriage return just before printing a prompt in the line editor.

`PROMPT_SUBST`
: If set, *parameter expansion*, *command substitution* and *arithmetic expansion* is performed in prompts.

`PUSHD_IGNORE_DUPS`
: Don't push multiple copies of the same directory onto the directory stack.

`PUSHD_MINUS`
: See Chapter 17 [Shell Builtin Commands], page 75, for the `popd` command.

`PUSHD_SILENT (-E)`
: Do not print the directory stack after `pushd` or `popd`.

`PUSHD_TO_HOME (-D)`
: Have `pushd` with no arguments act like `pushd $HOME`.

`RC_EXPAND_PARAM (-P)`
: Array expansions of the form `foo${xx}bar`, where the parameter `xx` is set to `(a b c)`, are substituted with `fooabar foobbar foocbar` instead of the default `fooa b cbar`.

`RC_QUOTES`
: Allow the character sequence `''` to signify a single quote within singly quoted strings.

`RCS (+f)`
: After '`/etc/zshenv`' is sourced on startup, source the '`/etc/zshrc`', '`.zshrc`', '`/etc/zlogin`', '`.zlogin`', and '`.zlogout`' files, as described in Chapter 4 [Startup/Shutdown Files], page 9. If this option is unset, only the '`/etc/zshenv`' file is sourced.

`REC_EXACT (-S)`
: In completion, recognize exact matches even if they are ambiguous.

`RM_STAR_SILENT (-H)`
: Do not query the user before executing `rm *` or `rm path/*`.

`SH_FILE_EXPANSION`
: Perform filename expansion (e.g., ~ expansion) before parameter expansion, command substitution, arithmetic expansion and brace expansion. If this option is unset, it is performed after brace expansion, so things like `~$USERNAME` and `~{pfalstad,rc}` will work.

`SH_GLOB`
: Disables the special meaning of `(`, `|`, `)` and `<` for globbing the result of parameter and command substitutions, and in some other places where the shell accepts patterns. This option is set if `zsh` is invoked as `sh` or `ksh`.

`SHIN_STDIN (-s, ksh: -s)`
: Commands are being read from the standard input. Commands are read from standard input if no command is specified with '`-c`' and no file of commands is specified. If `SHIN_STDIN` is set explicitly on the command line, any argument that would otherwise have been taken as a file to run will instead be treated as a normal positional parameter. Note that setting or un-setting this option on the command line does not necessarily affect the state the option will have while the shell is running; that is purely an indicator of whether or not commands are actually being read from standard input. The value of this option cannot be changed anywhere other than the command line.

`SH_OPTION_LETTERS`
: If this option is set the shell tries to interpret single letter options (which are used with `set` and `setopt`) like `ksh` does. This also affects the value of the `-` special parameter.

`SHORT_LOOPS`
: Allow the short forms of `for`, `select`, `if`, and `function` constructs.

`SH_WORD_SPLIT (-y)`
: See Section 6.3 [Parameter Expansion], page 16.

`SINGLE_COMMAND (-t, ksh: -t)`
: If the shell is reading from standard input, it exits after a single command has been executed. This also makes the shell non-interactive, unless the `INTERACTIVE` option is explicitly set on the command line. The value of this option cannot be changed anywhere other than the command line.

`SINGLE_LINE_ZLE (-M)`
: Use single-line command line editing instead of multi-line.

`SUN_KEYBOARD_HACK (-L)`
: If a line ends with a back-quote, and there are an odd number of back-quotes on the line, ignore the trailing back-quote. This is useful on some keyboards where the return key is too small, and the back-quote key lies annoyingly close to it.

`UNSET (+u, ksh: +u)`
: Treat unset parameters as if they were empty when substituting. Otherwise they are treated as an error.

`VERBOSE (-v, ksh: -v)`
: Print shell input lines as they are read.

`XTRACE (-x, ksh: -x)`
: Print commands and their arguments as they are executed.

`ZLE (-Z)`
: Use the zsh line editor.

16.3 Single Letter Options

16.3.1 Default Set

`-0`	`CORRECT`
`-1`	`PRINT_EXIT_VALUE`
`+2`	`BAD_PATTERN`
`+3`	`NOMATCH`
`-4`	`GLOB_DOTS`
`-5`	`NOTIFY`
`-6`	`BG_NICE`
`-7`	`IGNORE_EOF`
`-8`	`MARK_DIRS`
`-9`	`AUTO_LIST`
`+B`	`BEEP`
`+C`	`CLOBBER`
`-D`	`PUSHD_TO_HOME`
`-E`	`PUSHD_SILENT`
`+F`	`GLOB`
`-G`	`NULL_GLOB`

-H	RM_STAR_SILENT
-I	IGNORE_BRACES
-J	AUTO_CD
+K	BANG_HIST
-L	SUN_KEYBOARD_HACK
-M	SINGLE_LINE_ZLE
-N	AUTO_PUSHD
-O	CORRECT_ALL
-P	RC_EXPAND_PARAM
-Q	PATH_DIRS
-R	LONG_LIST_JOBS
-S	REC_EXACT
-T	CDABLE_VARS
-U	MAIL_WARNING
+V	PROMPT_CR
-W	AUTO_RESUME
-X	LIST_TYPES
-Y	MENU_COMPLETE
-Z	ZLE
-a	ALL_EXPORT
-e	ERR_EXIT
+f	RCS
-g	HIST_IGNORE_SPACE
-h	HIST_IGNORE_DUPS
-i	INTERACTIVE
-k	INTERACTIVE_COMMENTS
-l	LOGIN
-m	MONITOR
+n	EXEC
-p	PRIVILEGED
-s	SHIN_STDIN
-t	SINGLE_COMMAND
+u	UNSET
-v	VERBOSE
-w	CHASE_LINKS
-x	XTRACE
-y	SH_WORD_SPLIT

16.3.2 sh/ksh Emulation Set

`+C`	`CLOBBER`
`-X`	`MARK_DIRS`
`-a`	`ALL_EXPORT`
`-b`	`NOTIFY`
`-e`	`ERR_EXIT`
`+f`	`GLOB`
`-i`	`INTERACTIVE`
`-l`	`LOGIN`
`-m`	`MONITOR`
`+n`	`EXEC`
`-p`	`PRIVILEGED`
`-s`	`SHIN_STDIN`
`-t`	`SINGLE_COMMAND`
`+u`	`UNSET`
`-v`	`VERBOSE`
`-x`	`XTRACE`

16.3.3 Also Note

`-A`	Used by set for setting arrays
`-c`	Used on the command line to specify a single command
`-m`	Used by setopt for pattern-matching option setting
`-o`	Used in all places to allow use of long option names

Note that the use of '`-m`' in `setopt` and `unsetopt`, allowing the specification of option names by glob patterns, clashes with the use of '`-m`' for setting the `MONITOR` option.

17 Shell Builtin Commands

`- simple command`
: See Section 5.2 [Precommand Modifiers], page 11.

`.` *file* `[` *arg* ... `]`
: Read and execute commands from *file* and execute them in the current shell environment. If *file* does not contain a slash, or if `PATH_DIRS` is set, the shell looks in the components of `path` to find the directory containing *file*. Files in the current directory are not read unless '`.`' appears somewhere in `path`. If any arguments *arg* are given, they become the positional parameters; the old positional parameters are restored when the *file* is done executing. The exit status is the exit status of the last command executed.

`:` `[` *arg* ... `]`
: This command only expands parameters. A zero exit code is returned.

`alias [ -grmL ] [` *name*`[=`*value*`] ]` ...
: For each *name* with a corresponding *value*, define an alias with that value. A trailing space in *value* causes the next word to be checked for alias substitution. If the '`-g`' flag is present, define a global alias; global aliases are expanded even if they do not occur in command position. For each *name* with no *value*, print the value of *name*, if any. With no arguments, print all currently defined aliases. If the '`-m`' flag is given the arguments are taken as patterns (they should be quoted to preserve them from being interpreted as glob patterns) and the aliases matching these patterns are printed. When printing aliases and the '`-g`' or '`-r`' flags are present, then restrict the printing to global or regular aliases, respectively. If the '`-L`' flag is present, then print each alias in a manner suitable for putting in a startup script. The exit status is nonzero if a *name* (with no *value*) is given for which no alias has been defined.

`autoload [` *name* ... `]`
: For each of the *names* (which are names of functions), create a function marked undefined. The `fpath` variable will be searched to find the actual function definition when the function is first referenced. The definition is contained in a file of the same name as the function. If the file found contains a standard definition for the function, that is stored as the function; otherwise, the contents of the entire file are stored as the function. The latter format allows functions to be used directly as scripts.

`bg [` *job* ... `]`
job ... `&`
: Put each specified *job* in the background, or the current job if none is specified. See Chapter 10 [Jobs & Signals], page 35.

`bindkey -mevd`
`bindkey -r` *in-string* ...
`bindkey [ -a ]` *in-string* `[` *command* `]` ...
`bindkey -s [ -a ]` *in-string out-string* ...
: The '`-e`' and '`-v`' options put the keymaps in emacs mode and vi mode respectively; they cannot be used simultaneously. The '`-d`' option resets all bindings to the compiled-in settings. If not used with options '`-e`' or '`-v`', the maps will be left in emacs mode, or in vi mode if the `VISUAL` or `EDITOR` variables contain the string '`vi`'. Metafied characters are bound to self-insert by default. The '`-m`' option loads the compiled-in bindings of these characters for the mode determined by the preceding options, or the current mode if used alone. Any previous binding done by the user will be preserved. If the '`-r`' option is given, remove any binding for each *in-string*. If the '`-s`' option is not specified, bind each *in-string* to a specified *command*. If no *command* is specified, print the binding of *in-string* if it is bound, or return a nonzero exit code if it is not bound. If the '`-s`' option is specified, bind each *in-string* to each specified *out-string*.

When *in-string* is typed, *out-string* will be pushed back and treated as input to the line editor. The process is recursive, but to avoid infinite loops the shell will report an error if more than 20 consecutive replacements happen. If the '`-a`' option is specified, bind the *in-strings* in the alternative keymap instead of the standard one. The alternative keymap is used in vi command mode.

It's possible for an *in-string* to be bound to something and also be the beginning of a longer bound string. In this case the shell will wait a certain time to see if more characters are typed, and if not it will execute the binding. This timeout is defined by the `KEYTIMEOUT` parameter; the default is 0.4 seconds. No timeout is done if the prefix string is not bound.

For either *in-string* or *out-string*, control characters may be specified in the form `^X`, and the backslash may be used to introduce one of the following escape sequences:

`\a` Bell character

`\n` Linefeed (newline)

`\b` Backspace

`\t` Horizontal tab

`\v` Vertical tab

`\f` Form feed

`\r` Carriage return

`\e`
`\E` Escape

`\`*NNN* Character code in octal

`\x`*NN* Character code in hexadecimal

`\M-`*xxx* Character or escape sequence with meta bit set. The `-` after the `M` is optional.

`\C-X` Control character. The '`-`' after the `M` is optional.

In all other cases, `\` escapes the following character. Delete is written as `^?`. Note that `\M^?` and `^\M?` are not the same.

Multi-character *in-strings* cannot contain the null character (`^@` or `^ `). If they appear in a bindkey command, they will be silently translated to `\M-^@`. This restriction does not apply to *out-strings*, single-character *in-strings* and the first character of a multi-char *in-string*.

`break [` *n* `]`
Exit from an enclosing `for`, `while`, `until`, `select`, or `repeat` loop. If *n* is specified, then break *n* levels instead of just one.

`builtin` *name* `[` *args* `]` ...
Executes the builtin *name*, with the given *args*.

`bye` Same as `exit`.

`cd [` *arg* `]`
`cd` *old new*
`cd [+-]n` Change the current directory. In the first form, change the current directory to *arg*, or to the value of `HOME` if *arg* is not specified. If *arg* is `-`, change to the value of `OLDPWD`, the previous directory. If a directory named *arg* is not found in the current directory and *arg* does not begin with a slash, search each component of the shell parameter

cdpath. If the option CDABLEVARS is set, and a parameter named *arg* exists whose value begins with a slash, treat its value as the directory.

The second form of cd substitutes the string *new* for the string *old* in the name of the current directory, and tries to change to this new directory.

The third form of cd extracts an entry from the directory stack, and changes to that directory. An argument of the form +*n* identifies a stack entry by counting from the left of the list shown by the dirs command, starting with zero. An argument of the form -*n* counts from the right. If the PUSHD_MINUS option is set, the meanings of + and - in this context are swapped.

chdir
: Same as cd.

command simple command
: See Section 5.2 [Precommand Modifiers], page 11.

compctl
: See Chapter 18 [Programmable Completion], page 89.

continue [*num*]
: Resume the next iteration of the enclosing for, while, until, select, or repeat loop. If *n* is specified, break out of *n*-1 loops and resume at the *n*'th enclosing loop.

declare [*arg* ...]
: Same as typeset.

dirs [-v] [*arg* ...]
: With no arguments, print the contents of the directory stack. If the '-v' option is given, number the directories in the stack when printing. Directories are added to this stack with the pushd command, and removed with the cd or popd commands. If arguments are specified, load them onto the directory stack, replacing anything that was there, and push the current directory onto the stack.

disable [-afmr] *arg* ...
: Disable the hash table element named *arg* temporarily. The default is to disable builtin commands. This allows you to use an external command with the same name as a builtin command. The '-a' option causes disable to act on aliases. The '-f' option causes disable to act on shell functions. The '-r' option causes disable to act on reserved words. Without arguments all disabled hash table elements from the corresponding hash table are printed. With the '-m' flag the arguments are taken as patterns (which should be quoted to preserve them from being taken as glob patterns) and all hash table elements from the corresponding hash table matching these patterns are disabled. Disabled objects can be enabled with the enable command.

disown [*job* ...]
job ... &|
job ... &!
: Remove the specified jobs from the job table; the shell will no longer report their status, and will not complain if you try to exit an interactive shell with them running or stopped. If no job is specified use the current job.

echo [-neE] [*arg* ...]
: Write each *arg* on the standard output, with a space separating each one. If the '-n' flag is not present, print a newline at the end. echo recognizes the following escape sequences:

  \a   Bell

  \b   Backspace

  \c   Don't print an ending newline

  \e   Escape

`\f` Form feed

`\n` Newline

`\r` Carriage return

`\t` Horizontal tab

`\v` Vertical tab

`\\` Backslash

`\0NNN` Character code in octal, with a maximum of three digits after the zero. A non-octal digit terminates the number.

`\xNN` Character code in hexadecimal, with a maximum of two digits after the `x`. A non-hexadecimal digit terminates the number.

The '`-E`' flag or the `BSD_ECHO` option can be used to disable these escape sequences. In the later case '`-e`' flag can be used to enable them.

`echotc` *cap* `[` *arg* ... `]`
: Output the termcap string corresponding to the capability *cap*, with optional arguments.

`emulate [ -R ] [ zsh | sh | ksh | csh ]`
: Set up zsh options to emulate the specified shell as much as possible. `csh` will never be fully emulated. If the argument is not one of the shells listed above, `zsh` will be used as a default. If the '`-R`' option is given, all options are reset to their default value corresponding to the specified emulation mode.

`enable [ -afmr ]` *arg* ...
: Enable the hash table element named *arg*, presumably disabled earlier with `disable`. The default is to enable builtin commands. The '`-a`' option causes `enable` to act on aliases. The '`-f`' option causes `enable` to act on shell functions. The '`-r`' option causes `enable` to act on reserved words. Without arguments all enable hash table elements from the corresponding hash table are printed. With the '`-m`' flag the arguments are taken as patterns (should be quoted) and all hash table elements from the corresponding hash table matching these patterns are enabled. Enabled objects can be disabled with the `disable` builtin command.

`eval [` *arg* ... `]`
: Read the arguments as input to the shell and execute the resulting command(s) in the current shell process.

`exec simple command`
: See Section 5.2 [Precommand Modifiers], page 11.

`exit [` *n* `]`
: Exit the shell with the exit code specified by *n*; if none is specified, use the exit code from the last command executed. An `EOF` condition will also cause the shell to exit, unless the `IGNORE_EOF` option is set.

`export [` *name*`[=`*value*`]` ... `]`
: The specified *names* are marked for automatic export to the environment of subsequently executed commands. `export` is equivalent to `typeset -x`.

`false` Do nothing and return an exit code of 1.

`fc [ -e` *ename* `] [ -nlrdDfEim ] [` *old*`=`*new* ... `] [` *first* `[` *last* `]]`
`fc -ARWI [` *filename* `]`
: Select a range of commands from *first* to *last* from the history list. The arguments *first* and *last* may be specified as a number or as a string. A negative number is used

as an offset to the current history event number. A string specifies the most recent event beginning with the given string. All substitutions *old*=*new*, if any, are then performed on the commands. If the '-l' flag is given, the resulting commands are listed on standard output. If the '-m' flag is also given the first argument is taken as a pattern (which should be quoted), and only the history events matching this pattern will be shown. Otherwise the editor program *ename* is invoked on a file containing these history events. If *ename* is not given, the value of the parameter `FCEDIT` is used. If *ename* is `-`, no editor is invoked. When editing is complete, the edited command(s) is executed. If *first* is not specified, it will be set to -1 (the most recent event), or to -16 if the '-l' flag is given. If *last* is not specified, it will be set to *first*, or to -1 if the '-l' flag is given. The flag '-r' reverses the order of the commands and the flag '-n' suppresses command numbers when listing. Also when listing, '-d' prints timestamps for each command, '-f' prints full time and date stamps. Adding the '-E' flag causes the dates to be printed as (`dd.mm.yyyy`), instead of the default, `mm/dd/yyyy`. Adding the '-i' flag causes the dates to be printed as `yyyy-mm-dd`, in a fixed format. With the '-D' flag, `fc` prints elapsed times.

`fc -R` reads the history from the given file, `fc -W` writes the history out to the given file, and `fc -A` appends the history out to the given file. `fc -AI` (`WI`) appends (writes) only those events that are new since the last incremental append (write) to the history file. In any case the file will have no more than `SAVEHIST` entries.

`fg` [*job* ...]
job ... Bring the specified *job*s to the foreground. If no *job* is specified, use the current job.

`functions [ +-tum ] [` *name* ...]
Equivalent to `typeset -f`.

`getln` *name* ...
Read the top value from the buffer stack and put it in the shell parameter *name*. Equivalent to `read -zr`. The flags '-c', '-l', '-A', '-e', '-E', and '-n' are also supported.

`getopts` *optstring name* [*arg* ...]
Checks *arg* for legal options. If *arg* is omitted, use the positional parameters. A valid option argument begins with a `+` or a `-`. An argument not beginning with a `+` or a `-`, or the argument `--`, ends the options. *optstring* contains the letters that `getopts` recognizes. If a letter is followed by a `:`, that option is expected to have an argument. The options can be separated from the argument by blanks.

Each time it is invoked, `getopts` places the option letter it finds in the shell parameter *name*, prepended with a `+` when *arg* begins with a `+`. The index of the next *arg* is stored in `OPTIND`. The option argument, if any, is stored in `OPTARG`.

A leading `:` in *optstring* causes `getopts` to store the letter of the invalid option in `OPTARG`, and to set *name* to `?` for an unknown option and to `:` when a required option is missing. Otherwise, `getopts` prints an error message. The exit status is nonzero when there are no more options.

`hash [ -dfmr ] [` *name*[=*value*]] ...
With no arguments or options, `hash` will list the entire command hash table.

The '-m' option causes the arguments to be taken as patterns (they should be quoted) and the elements of the command hash table matching these patterns are printed.

The '-r' option causes the command hash table to be thrown out and restarted. The '-f' option causes the entire path to be searched, and all the commands found are added to the hash table. These options cannot be used with any arguments.

For each *name* with a corresponding *value*, put *name* in the command hash table, associating it with the pathname *value*. Whenever *name* is used as a command argument, the shell will try to execute the file given by *value*. For each *name* with no corresponding *value*, search for *name* in the path, and add it to the command hash table, and associating it with the discovered path, if it is found.

Adding the '`-d`' option causes `hash` to act on the named directory table instead of the command hash table. The remaining discussion of `hash` will assume that the '`-d`' is given.

If invoked without any arguments, and without any other options, `hash -d` lists the entire named directory table.

The '`-m`' option causes the arguments to be taken as patterns (they should be quoted) and the elements of the named directory table matching these patterns are printed.

The '`-r`' option causes the named directory table to be thrown out and restarted so that it only contains `~`. The '`-f`' option causes all usernames to be added to the named directory table. These options cannot be used with any arguments.

For each *name* with a corresponding *value*, put *name* in the named directory table. The directory name *name* is then associated with the specified path *value*, so that *value* may be referred to as `~`*name*. For each *name* with no corresponding *value*, search for as a username and as a parameter. If it is found, it is added to the named directory hash table.

`history [ -nrdDfEim ] [` *first* `[` *last* `]]`
: Same as `fc -l`.

`integer [ +-lrtux ] [` *name*`[=`*value*`] ]`
: Same as `typeset -i`, except that options irrelevant to integers are not permitted.

`jobs [ -lprs ] [` *job* ... `]`
: Lists information about each given job, or all jobs if *job* is omitted. The '`-l`' flag lists process ids, and the '`-p`' flag lists process groups. If the '`-r`' flag is given only running jobs will be listed; if the '`-s`' flag is given only stopped jobs are shown.

`kill [ -s` *signal'name* `]` *job* ...
`kill [ -`*sig* `]` *job* ...
`kill -l [` *sig* ... `]`
: Sends either `SIGTERM` or the specified signal to the given jobs or processes. Signals are given by number or by names, without the `SIG` prefix. If the signal being sent is not `KILL` or `CONT`, then the job will be sent a `CONT` signal if it is stopped. The argument *job* can be the process id of a job not in the job list. In the third form, `kill -l`, if *sig* is not specified the signal names are listed. Otherwise, for each *sig* that is a name, the corresponding signal number is listed. For each *sig* that is a signal number or a number representing the exit status of a process which was terminated or stopped by a signal the name of the signal is printed.

`let` *arg* ...
: Evaluate each *arg* as an arithmetic expression. See Chapter 11 [Arithmetic Evaluation], page 37, for a description of arithmetic expressions. The exit status is 0 if the value of the last expression is nonzero, and 1 otherwise.

`limit [ -hs ] [` *resource* `[` *limit* `] ]` ...
: Set or display resource limits. Unless the '`-s`' flag is given the limit applies only the children of the shell. If '`-s`' is given without other arguments, the resource limits of the current shell is set to the previously set resource limits of the children. If *limit* is not specified, print the current limit placed on *resource*; otherwise set the limit to the specified value. If the '`-h`' flag is given, use hard limits instead of soft limits. If no *resource* is given, print all limits.
resource is one of:

`cputime` Maximum CPU seconds per process.

`filesize` Largest single file allowed.

`datasize` Maximum data size (including stack) for each process.

`stacksize`
: Maximum stack size for each process.

`coredumpsize`
: Maximum size of a core dump.

`resident`
`memoryuse`
: Maximum resident set size.

`memorylocked`
: Maximum amount of memory locked in RAM.

`descriptors`
: Maximum value for a file descriptor.

`openfiles`
: Maximum number of open files.

`vmemorysize`
: Maximum amount of virtual memory.

Which of these resource limits are available depends on the system. *limit* is a number, with an optional scaling factor, as follows:

n`h` Hours.

n`k` Kilobytes. This is the default for all but cputime.

n`m` Megabytes or minutes.

mm`:`*ss* Minutes and seconds.

`local [ +-LRZilrtu [`*n*`]] [ `*name*`[=`*value*`] ]`
: Same as `typeset`, except that the options '`-x`' and '`-f`' are not permitted.

`log` List all users currently logged in who are affected by the current setting of the `watch` parameter.

`logout` Exit the shell, if this is a login shell.

`noglob simple command`
: See Section 5.2 [Precommand Modifiers], page 11.

`popd [ +-`*n* `]`
: Removes a entry from the directory stack and, performs a `cd` to the new top directory. With no argument, the current top entry is removed. An argument of the form `+`*n* identifies a stack entry by counting from the left of the list shown by the `dirs` command, starting with zero. An argument of the form '`-n`' counts from the right. If the `PUSHD_MINUS` option is set, the meanings of `+` and `-` in this context are swapped.

`print [ -nrslzpNDPoOicm ] [ -u`*n* `] [ -R [ -en ]] [ `*arg* `... ]`
: With no flags or with flag `-`, the arguments are printed on the standard output as described by `echo`, with the following differences: the escape sequence `\M-x` metafies the character `x` (sets the highest bit), `\C-x` produces a control character (`\C-@` and `\C-?` give the characters NULL and delete) and `\E` is a synonym for `\e`. Finally, if not in an escape sequence, `\` escapes the following character and is not printed.

`-r` Ignore the escape conventions of `echo`.

`-R` Emulate the BSD `echo` command which does not process escape sequences unless the '`-e`' flag is given. The '`-n`' flag suppresses the trailing newline. Only the '`-e`' and '`-n`' flags are recognized after '`-R`', all other arguments and options are printed.

`-m` Take the fist argument as a pattern (should be quoted) and remove it from the argument list together with subsequent arguments that do not match this pattern.

`-s` Place the results in the history list instead of on the standard output.

`-n` Do not add a newline to the output.

`-l` Print the arguments separated by newlines instead of spaces.

`-N` Print the arguments separated and terminated by nulls.

`-o` Print the arguments sorted in ascending order.

`-O` Print the arguments sorted in descending order.

`-i` If given together with '`-o`' or '`-O`', makes the sort be case-insensitive.

`-c` Print the arguments in columns.

`-u`*n* Print the arguments to file descriptor *n*.

`-p` Print the arguments to the input of the coprocess.

`-z` Push the arguments onto the editing buffer stack, separated by spaces; no escape sequences are recognized.

`-D` Treat the arguments as directory names, replacing prefixes with `~` expressions, as appropriate.

`-P` Recognize the same escape sequences as in the `PROMPT` parameter.

`pushd [` *arg* `]`
`pushd` *old new*
`pushd +-`*n* Change the current directory, and push the old current directory onto the directory stack. In the first form, change the current directory to *arg*. If *arg* is not specified, change to the second directory on the stack (that is, exchange the top two entries), or change to the value of `HOME` if the `PUSHD_TO_HOME` option is set or if there is only one entry on the stack. If *arg* is `-`, change to the value of `OLDPWD`, the previous directory. If a directory named *arg* is not found in the current directory and *arg* does not contain a slash, search each component of the shell parameter `cdpath`. If the option `CDABLEVARS` is set, and a parameter named *arg* exists whose value begins with a slash, treat its value as the directory. If the option `PUSHD_SILENT` is not set, the directory stack will be printed after a `pushd` is performed.

The second form of `pushd` substitutes the string *new* for the string *old* in the name of the current directory, and tries to change to this new directory.

The third form of `pushd` changes directory by rotating the directory list. An argument of the form `+n` identifies a stack entry by counting from the left of the list shown by the `dirs` command, starting with zero. An argument of the form '`-n`' counts from the right. If the `PUSHD_MINUS` option is set, the meanings of `+` and `-` in this context are swapped.

`pushln` Equivalent to `print -nz`.

`pwd [ -r ]`
Print the absolute pathname of the current working directory. If the '`-r`' flag is specified or the `CHASE_LINKS` option is set, the printed path will not contain symbolic links.

`r` Equivalent to `fc -e -`.

`read [ -rzpqAclneE ] [ -k [`*num*`] ] [ -u`*n* `] [` *name?prompt* `] [` *name* ... `]`
Read one line and break it into fields using the characters in `IFS` as separators.

`-r` Raw mode: a `\` at the end of a line does not signify line continuation.

`-q` Read only one character from the terminal and set *name* to '`y`' if this character was '`y`' or '`Y`' and to '`n`' otherwise. With this flag set the return value is zero only if the character was '`y`' or '`Y`'.

`-k [` *num* `]`
Read only one (or *num*) characters from the terminal.

`-z` Read from the editor buffer stack. The first field is assigned to the first *name*, the second field to the second *name*, etc., with leftover fields assigned to the last *name*.

`-e`
`-E` The words read are printed after the whole line is read. If the '`-e`' flag is set, the words are not assigned to the parameters.

`-A` The first *name* is taken as the name of an array and all words are assigned to it.

`-c`
`-l` These flags are allowed only if called inside a function used for completion (specified with the '`-K`' flag to `compctl`). If the '`-c`' flag is given, the words of the current command are read. If the '`-l`' flag is given, the whole line is assigned as a scalar. If `name` is omitted then `REPLY` is used for scalars and `reply` for arrays.

`-n` Together with either of the previous flags, this option gives the number of the word the cursor is on or the index of the character the cursor is on respectively.

`-u`*n* Input is read from file descriptor *n*.

`-p` Input is read from the coprocess.

If the first argument contains a `?`, the remainder of this word is used as a `prompt` on standard error when the shell is interactive. The exit status is `0` unless an end-of-file is encountered.

`readonly [` *name*`[=`*value*`]] ...`
The given *names* are marked readonly; these names cannot be changed by subsequent assignment.

`rehash [ -df ]`
Throw out the command hash table and start over. If the '`-f`' option is set, rescan the command path immediately, instead of rebuilding the hash table incrementally.

The '`-d`' option causes `rehash` to act on the named directory table instead of the command hash table. This reduces the named directory table to only the `~` entry. If the '`-f`' option is also used, the named directory table is rebuilt immediately.

`rehash` is equivalent to `hash -r`.

`return [` *n* `]`
Causes a shell function or `.` script to return to the invoking script with the return status specified by *n*. If *n* is omitted then the return status is that of the last command executed.

If `return` was executed from a trap in a '`TRAPxxx`' function, the effect is different for zero and nonzero return status. With zero status (or after an implicit return at the end of the trap), the shell will return to whatever it was previously processing; with a non-zero status, the shell will behave as interrupted except that the return status of the trap is retained. Note that the signal which caused the trap is passed as the first argument, so the statement '`return $((128+$1))`' will return the same status as if the signal had not been trapped.

`sched [+]`*hh*`:`*mm command* ...
`sched [ -`*item* `]`
: Make an entry in the scheduled list of commands to execute. The time may be specified in either absolute or relative time. With no arguments, prints the list of scheduled commands. With the argument `-`*item*, removes the given item from the list.

`set [ +-`*options* `] [ +-o` *option name* `]` ... `[ +-A [` *name* `] ] [` *arg* ... `]`
: Set the options for the shell and/or set the positional parameters, or declare an array. If the '`-s`' option is given it causes the specified arguments to be sorted before assigning them to the positional parameters (or to the array *name* if '`-A`' is used). With '`+s`' sort arguments in descending order. See Chapter 16 [Options], page 65, for the meaning of the other flags. Flags may be specified by name using the '`-o`' option. If the '`-A`' flag is specified, *name* is set to an array containing the given *args*; if '`+A`' is used and *name* is an array, the gien arguments will replace the initial elements of that array; if no *name* is specified, all arrays are printed. Otherwise the positional parameters are set. If no arguments are given, then the names and values of all parameters are printed on the standard output. If the only argument is `+`, the names of all parameters are printed.

`setopt [ -m ] [ +-`*options* `] [` *name* ... `]`
: Set the options for the shell. All options specified either with flags or by name are set. If no arguments are supplied, the names of all options currently set are printed. In option names, case is insignificant, and all underscore characters are ignored. If the '`-m`' flag is given the arguments are taken as patterns (which should be quoted to preserve them from being interpreted as glob patterns), and all options with names matching these patterns are set.

`shift [` *n* `] [` *name* ... `]`
: The positional parameters from `$`*n*`+1` ... are renamed `$1`, where *n* is an arithmetic expression that defaults to 1. If any *names* are given then the arrays with these names are shifted, instead of the positional parameters.

`source`
: Same as '`.`', except that the current directory is always searched and is always searched first, before directories in `path`.

`suspend [ -f ]`
: Suspend the execution of the shell (send it a `SIGTSTP`) until it receives a `SIGCONT`. If the '`-f`' option is not given, complain if this is a login shell.

`test` *arg* ...
`[` *arg* ... `]`
: Like the system version of `test`. Added for compatibility; use conditional expressions instead.

`times`
: Print the accumulated user and system times for the shell and for processes run from the shell.

`trap [` *arg* `] [` *sig* `]` ...
: *arg* is a command to be read and executed when the shell receives *sig*. Each *sig* can be given as a number or as the name of a signal. If *arg* is `-`, then all traps *sig* are reset to their default values. If *arg* is the null string, then this signal is ignored by the shell and by the commands it invokes. If *sig* is `ZERR` then *arg* will be executed after each

command with a nonzero exit status. If *sig* is `DEBUG` then *arg* will be executed after each command. If *sig* is `0` or `EXIT` and the `trap` statement is executed inside the body of a function, then the command *arg* is executed after the function completes. If *sig* is `0` or `EXIT` and the `trap` statement is not executed inside the body of a function, then the command *arg* is executed when the shell terminates. The `trap` command with no arguments prints a list of commands associated with each signal.

`true` Do nothing and return an exit code of 0.

`ttyctl [ -fu ]`
The '`-f`' option freezes the tty, and '`-u`' un-freezes it. When the tty is frozen, no changes made to the tty settings by external programs will be honoured by the shell, except for changes in the size of the screen; the shell will simply reset the settings to their previous values as soon as each command exits. Thus, `stty` and similar programs have no effect when the tty is frozen. Without options it reports whether the terminal is frozen or not.

`type [ -fpam ]` *name* ...
Same as `whence -v`.

`typeset [ +-LRUZfilrtuxm [`*n*`]] [` *name*`[=`*value*`]] ...`
Set attributes and values for shell parameters. When invoked inside a function, a new parameter is created which will be unset when the function completes. The new parameter will not be exported unless `ALL_EXPORT` is set, in which case the parameter will be exported provided no parameter of that name already exists. The following attributes are valid:

`-L` Left justify and remove leading blanks from *value*. If *n* is nonzero, it defines the width of the field; otherwise it is determined by the width of the value of the first assignment. When the parameter is printed, it is filled on the right with blanks or truncated if necessary to fit the field. Leading zeros are removed if the '`-Z`' flag is also set.

`-R` Right justify and fill with leading blanks. If *n* is nonzero it defines the width of the field; otherwise it is determined by the width of the value of the first assignment. When the parameter is printed, the field is left filled with blanks or truncated from the end.

`-U` For arrays keep only the first element of each duplications. It can also be set for colon separated special parameters like `PATH` or `FIGNORE`, etc.

`-Z` Right justify and fill with leading zeros if the first non-blank character is a digit and the '`-L`' flag has not been set. If *n* is nonzero it defines the width of the field; otherwise it is determined by the width of the value of the first assignment.

`-f` The names refer to functions rather than parameters. No assignments can be made, and the only other valid flags are '`-t`' and '`-u`'. The flag '`-t`' turns on execution tracing for this function. The flag '`-u`' causes this function to be marked for autoloading. The `fpath` parameter will be searched to find the function definition when the function is first referenced.; see `autoload`.

`-i` Use an internal integer representation. If *n* is nonzero it defines the output arithmetic base, otherwise it is determined by the first assignment.

`-l` Convert to lower case.

`-r` The given *names* are marked read-only.

`-t` Tags the named parameters. Tags have no special meaning to the shell.

`-u` Convert to upper case.

`-x` Mark for automatic export to the environment of subsequently executed commands.

Using `+` rather than `-` causes these flags to be turned off. If no arguments are given but flags are specified, a list of named parameters which have these flags set is printed. Using `+` instead of `-` keeps their values from being printed. If no arguments or options are given, the names and attributes of all parameters are printed. If only the '`-m`' flag is given the arguments are taken as patterns (which should be quoted), and all parameters or functions (with the '`-f`' flag) with matching names are printed.

`ulimit [ -SHacdflmnpstv ] [` *limit* `]` ...
Set or display resource limits of the shell and the processes started by the shell. The value of *limit* can be a number in the unit specified below or the value `unlimited`. If the '`-H`' flag is given use hard limits instead of soft limits. If the '`-S`' flag is given together with the '`-H`' flag set both hard and soft limits. If no options are used, the file size limit ('`-f`') is assumed. If *limit* is omitted the current value of the specified resources are printed. When more than one resource values are printed the limit name and unit is printed before each value.

`-a` Lists all of the current resource limits.

`-c` Maximum size of core dumps, in 512-byte blocks.

`-d` Maximum size of the data segment, in Kbytes.

`-f` Maximum size of individual files written, in 512-byte blocks.

`-l` Maximum size of locked-in memory, in Kbytes.

`-m` Maximum size of physical memory, in Kbytes.

`-n` Maximum number of open file descriptors.

`-s` Maximum size of stack, in Kbytes.

`-t` Maximum number of CPU seconds.

`-u` The number of processes available to the user.

`-v` Maximum size of virtual memory, in Kbytes.

`umask [ -S] [` *mask* `]`
The umask is set to *mask*. *mask* can be either an octal number or a symbolic value as described in `chmod(1)`. If *mask* is omitted, the current value is printed. The '`-S`' option causes the mask to be printed as a symbolic value. Otherwise, the mask is printed as an octal number. Note that in the symbolic form the permissions you specify are those which are to be allowed (not denied) to the users specified).

`unalias [ -m ]` *name* ...
The alias definition, if any, for each *name* is removed. With the '`-m`' flag, the arguments are taken as patterns (which should be quoted), and all aliases with matching names are removed. `unalias` is equivalent to `unhash -a`.

`unfunction [ -m ]` *name* ...
The function definition, if any, for each *name* is removed. With the '`-m`' flag, the arguments are taken as patterns (which should be quoted), and all function with matching names are removed. `unfunction` is equivalent to `unhash -f`.

`unhash [ -adfm ]` *name* ...
Remove the element named *name* from an internal hash table. The default is remove elements from the command hash table. The '`-a`' option causes `unhash` to remove

aliases. The '-f' option causes `unhash` to remove shell functions. The '-d' options causes `unhash` to remove named directories. If the '-m' flag is given the arguments are taken as patterns (should be quoted) and all elements of the corresponding hash table with matching names will be removed.

`unlimit [ -hs ]` *resource ...*
: The resource limit for each *resource* is set to the hard limit. If the '-h' flag is given and the shell is running as root, the hard resource limit for each *resource* is removed. The resources of the shell process are only changed if the '-s' flag is given.

`unset [ -fm ]` *name ...*
: Each named parameter is unset. If the '-m' flag is set, the arguments are taken as patterns (which should be quoted), and all parameters with matching names are unset. `Unset -f` is equivalent to `unfunction`.

`unsetopt [ -m ] [ +-`*options* `] [` *name ...* `]`
: Unset the options for the shell. All options specified either with flags or by name are unset. If the '-m' flag is given, the arguments are taken as patterns (which should be quoted), and all options with names matching these patterns are unset.

`vared [ -c ] [ -h ] [ -p` *prompt*`] [ -r` *rprompt* `]` *name*
: The value of the parameter *name* is loaded into the edit buffer, and the line editor is invoked. When the editor exits, *name* is set to the string value returned by the editor. If the '-c' flag is given, the parameter is created if it doesn't already exist. If the '-p' flag is given, *prompt* will be taken as the prompt to display at the left and if the '-r' flag is given, the following string gives the prompt to display at the right. If the '-h' flag is specified, the history can be accessed from `zle`.

`wait [` *job ...* `]`
: Wait for the specified jobs or processes. If *job* is not given then all currently active child processes are waited for. Each *job* can be either a job specification or the process-id of a job in the job table. The exit status from this command is that of the job waited for.

`whence [ -vcfpam ]` *name ...*
: For each name, indicate how it would be interpreted if used as a command name. The '-v' flag produces a more verbose report. The '-c' flag prints the results in a csh-like format and takes precedence over '-v'. The '-f' flag causes the contents of a shell function to be displayed, which would otherwise not happen unless the '-c' flag were used. The '-p' flag does a path search for *name* even if it is an alias, reserved word, shell function or builtin. The '-a' flag does a search for all occurrences of *name* throughout the command path. With the '-m' flag, the arguments are taken as patterns (which should be quoted), and the information is displayed for each command matching one of these patterns.

`where`
: Same as `whence -ca`.

`which [ -pam ]` *name ...*
: Same as `whence -c`.

18 Programmable Completion

`compctl` [`-CDT`] *options* [*command* ...]

`compctl` [`-CDT`] *options*
 [`-x` *pattern options* `-` ... `--`] [`+` *options* [`-x` ... `--`] ... [`+`]]
 [*command* ...]

`compctl -L` [`-CDT`] [*command* ...]

`compctl +` *command* ...

Control the editor's completion behaviour according to the supplied set of *options*. Various editing commands, notably `expand-or-complete-word`, usually bound to ⟨TAB⟩, will attempt to complete a word typed by the user, while others, notably `delete-char-or-list`, usually bound to `^D` in emacs editing mode, list the possibilities; `compctl` controls what those possibilities are. They may for example be filenames (the most common case, and hence the default), shell variables, or words from a user-specified list.

18.1 Command Flags

Completion of the arguments of a command may be different for each command or may use the default. The behaviour when completing the command word itself may also be separately specified. These correspond to the following flags and arguments, all of which (except for '`-L`') may be combined with any combination of the options described subsequently in Section 18.2 [Options Flags], page 90.

command ...
: controls completion for the named commands, which must be listed last on the command line. If completion is attempted for a command with a pathname containing slashes and no completion definition is found, the search is retried with the last pathname component. Note that aliases are expanded before the command name is determined unless the `COMPLETE_ALIASES` option is set. Commands should not be combined with the '`-D`', '`-C`' or '`-T`' flags.

`-D`
: controls default completion behavior for the arguments of commands not assigned any special behavior. If no `compctl -D` command has been issued, filenames are completed.

`-C`
: controls completion when the command word itself is being completed. If no `compctl -C` command has been issued, the names of any executable command (whether in the path or specific to the shell, such as aliases or functions) are completed.

`-T`
: supplies completion flags to be used before any other processing is done, even those given to specific commands with other `compctl` definitions. This is only useful when combined with extended completion (the '`-x`' flag. See Section 18.4 [Extended Completion], page 93). Using this flag you can define default behaviour which will apply to all commands without exception, or you can alter the standard behaviour for all commands. For example, if your access to the user database is too slow and/or it contains too many users (so that completion after `~` is too slow to be usable), you can use

  `compctl -Tx 'C[0,*/*]' -f - 's[~]' -k` *friends* `-S/`

to complete the strings in the array *friends* after a ~. The first argument is necessary so that this form of ~-completion is not tried after the directory name is finished.

`-L` lists the existing completion behaviour in a manner suitable for putting into a start-up script; the existing behaviour is not changed. Any combination of the above forms may be specified, otherwise all defined completions are listed. Any other flags supplied are ignored.

`no argument`
If no argument is given, `compctl` lists all defined completions in an abbreviated form; with a list of *options*, all completions with those flags set (not counting extended completion) are listed.

If the `+` flag is alone and followed immediately by the *command* list, the completion behaviour for all the commands in the list is reset to the default. In other words, completion will subsequently use the options specified by the '`-D`' flag.

18.2 Options Flags

`[ -fcFBdeaRGovNAIOPZEnbjrzu ]`
`[ -k` *array* `] [ -g` *globstring* `] [ -s` *subststring* `]`
`[ -K` *function* `] [ -H` *num pattern* `]`
`[ -Q ] [ -P` *prefix* `] [ -S` *suffix* `]`
`[ -q ] [ -X` *explanation* `]`
`[ -l` *cmd* `] [ -U ]`

The remaining options specify the type of command arguments to look for during completion. Any combination of these flags may be specified; the result is a sorted list of all the possibilities. The options are described in the following sections.

18.2.1 Simple Flags

These produce completion lists made up by the shell itself:

`-f` Filenames and file-system paths.

`-c` Command names, including aliases, shell functions, builtins and reserved words.

`-F` Function names.

`-B` Names of builtin commands.

`-m` Names of external commands.

`-w` Reserved words.

`-a` Alias names.

`-R` Names of regular (non-global) aliases.

`-G` Names of global aliases.

`-d` This can be combined with '`-F`', '`-B`', '`-w`', '`-a`', '`-R`' and '`-G`' to get names of disabled functions, builtins, reserved words or aliases.

`-e` This option (to show enabled commands) is in effect by default, but may be combined with '`-d`'; '`-de`' in combination with '`-F`', '`-B`', '`-w`', '`-a`', '`-R`' and '`-G`' will complete names of functions, builtins, reserved words or aliases whether or not they are disabled.

`-o` Names of shell options. See Chapter 16 [Options], page 65.

`-v` Names of any variable defined in the shell.

`-N` Names of scalar (non-array) parameters.

`-A` Array names.

`-I` Names of integer variables.

`-O` Names of read-only variables.

`-p` Names of parameters used by the shell (including special parameters).

`-Z` Names of shell special parameters.

`-E` Names of environment variables.

`-n` Named directories.

`-b` Key binding names.

`-j` Job names: the first word of the job leader's command line. This is useful with the kill builtin.

`-r` Names of running jobs.

`-z` Names of suspended jobs.

`-u` User names.

18.2.2 Flags with arguments

These have user supplied arguments to determine how the list of completions is to be made up:

`-k` *array* Names taken from the elements of `$`*array* (note that the `$` does not appear on the command line). Alternatively, the argument *array* itself may be a set of space or comma separated values in parentheses, in which any delimiter may be escaped with a backslash; in this case the argument should be quoted. For example, '`compctl -k "(cputime filesize datasize stacksize coredumpsize resident descriptors)" limit`'.

`-g` *globstring*

The *globstring* is expanded using filename globbing; it should be quoted to protect it from immediate expansion. The resulting filenames are taken as possible completions. Use `*(/)` instead of `*/` for directories. The `fignore` special parameter is not applied to the resulting files. More than one pattern may be given separated by blanks. (Note that brace expansion is not part of globbing. Use the syntax `(either|or)` to match alternatives.)

`-K` *function*

Call the given function to get the completions. The function is passed two arguments: the prefix and the suffix of the word on which completion is to be attempted, in other words those characters before the cursor position, and those from the cursor position onwards. The function should set the variable `reply` to an array containing the completions (one completion per element); note that `reply` should not be made local to the function. From such a function the command line can be accessed with the '`-c`' and '`-l`' flags to the `read` builtin. For example,

```
function whoson { reply=(`users`); }
compctl -K whoson talk
```

completes only logged-on users after '`talk`'. Note that `whoson` must return an array so that just ``reply=`users` `` is incorrect.

`-H` *num pattern*
: The possible completions are taken from the last *num* history lines. Only words matching *pattern* are taken. If *num* is zero or negative the whole history is searched and if *pattern* is the empty string all words are taken (as with `*`). A typical use is

```
compctl -D -f + -H 0 '' -X '(No file found; using history)'
```

which forces completion to look back in the history list for a word if no filename matches. The explanation string is useful as it tells the user that no file of that name exists, which is otherwise ambiguous. (See the next section for '`-X`'.)

18.2.3 Control Flags

These do not directly specify types of name to be completed, but manipulate the options that do:

`-Q`
: This instructs the shell not to quote any metacharacters in the possible completions. Normally the results of a completion are inserted into the command line with any metacharacters quoted so that they are interpreted as normal characters. This is appropriate for filenames and ordinary strings. However, for special effects, such as inserting a backquoted expression from a completion array ('`-k`') so that the expression will not be evaluated until the complete line is executed, this option must be used.

`-P` *prefix*
: The *prefix* is inserted just before the completed string; any initial part already typed will be completed and the whole *prefix* ignored for completion purposes. For example,

```
compctl -j -P "%" kill
```

inserts a `%` after the `kill` command and then completes job names.

`-S` *suffix*
: When a completion is found the *suffix* is inserted after the completed string. In the case of menu completion the *suffix* is inserted immediately, but it is still possible to cycle through the list of completions by repeatedly hitting the same key.

`-q`
: If used with a suffix as specified by the previous option, this causes the suffix to be removed if the next character typed is a blank or does not insert anything (the same rule as used for the `AUTO_REMOVE_SLASH` option). The option is most useful for list separators (comma, colon, etc.).

`-l` *cmd*
: This option cannot be combined with any other option. It restricts the range of command line words that are considered to be arguments. If combined with one of the extended completion patterns '`p[...]`', '`r[...]`', or '`R[...]`' (See Section 18.4 [Extended Completion], page 93.) the range is restricted to the arguments specified in the brackets. Completion is then performed as if these had been given as arguments to the *cmd* supplied with the option. If the *cmd* string is empty the first word in the range is instead taken as the command name, and command name completion performed on the first word in the range. For example,

```
compctl -x 'r[-exec,;]' -l '' -- find
```

completes arguments between `-exec` and the following `;` (or the end of the command line if there is no such string) as if they were a separate command line.

`-U`
: Use the whole list of possible completions, whether or not they actually match the word on the command line. The word typed so far will be deleted. This is most useful with a function (given by the '`-K`' option) which can examine the word components passed to it (or via the `read` builtin's '`-c`' and '`-l`' flags) and use its own criteria to decide what matches. If there is no completion, the original word is retained.

`-X` *explanation*
: Print *explanation* when trying completion on the current set of options. A `%n` in this string is replaced by the number of matches.

18.3 Alternative Completion

`compctl` [`-CDT`] *options* `+` *options* [`+` ...] [`+`] *command* ...

The form with `+` specifies alternative *options*. Completion is tried with the *options* before the first `+`. If this produces no matches completion is tried with the flags after the `+` and so on. If there are no flags after the last `+` and a match has not been found up to that point, default completion is tried.

18.4 Extended Completion

`compctl` [`-CDT`] *options* `-x` *pattern options* `-` ... `--` [*command* ...]

`compctl` [`-CDT`] *options* [`-x` *pattern options* `-` ... `--`]
[`+` *options* [`-x` ... `--`] ... [`+`]] [*command* ...]

The form with '`-x`' specifies extended completion for the commands given; as shown, it may be combined with alternative completion using `+`. Each *pattern* is examined in turn; when a match is found, the corresponding *options*, as described in Section 18.2 [Options Flags], page 90, are used to generate possible completions. If no *pattern* matches, the *options* given before the '`-x`' are used.

Note that each pattern should be supplied as a single argument and should be quoted to prevent expansion of meta-characters by the shell.

A *pattern* is built of sub-patterns separated by commas; it matches if at least one of these sub-patterns matches (they are `or`'ed). These sub-patterns are in turn composed of other sub-patterns separated by white spaces which match if all of the sub-patterns match (they are `and`'ed). An element of the sub-patterns is of the form `c[...][...]`, where the pairs of brackets may be repeated as often as necessary, and matches if any of the sets of brackets match (an `or`). The example below makes this clearer.

The elements may be any of the following:

s`[`*string*`]` ...
: Matches if the current word on the command line starts with one of the strings given in brackets. The *string* is not removed and is not part of the completion.

S`[`*string*`]` ...
: Like *s*[*string*] except that the *string* is part of the completion.

p`[`*from*`,`*to*`]` ...
: Matches if the number of the current word is between one of the *from* and *to* pairs inclusive. The comma and *to* are optional; *to* defaults to the same value as *from*. The numbers may be negative: '`-n`' refers to the `n`'th last word on the line.

c`[`*offset*`,`*string*`]` ...
: Matches if the *string* matches the word offset by *offset* from the current word position. Usually *offset* will be negative.

C`[`*offset*`,`*pattern*`]` ...
: Like *c* but using pattern matching instead.

w[*index*,*string*] ...
: Matches if the word in position *index* is equal to the corresponding *string*. Note that the word count is made after any alias expansion.

W[*index*,*pattern*] ...
: Like *w* but using pattern matching instead.

n[*index*,*string*] ...
: Matches if the current word contains *string*. Anything up to and including the *index*'th occurrence of this *string* will not be considered part of the completion, but the rest will. *index* may be negative to count from the end: in most cases, *index* will be 1 or -1.

N[*index*,*string*] ...
: Like *n*[*index*,*string*] except that the *string* will be taken as a character class. Anything up to and including the *index*'th occurrence of any of the characters in *string* will not be considered part of the completion.

m[*min*,*max*] ...
: Matches if the total number of words lies between *min* and *max* inclusive.

r[*str1*,*str2*] ...
: Matches if the cursor is after a word with prefix *str1*. If there is also a word with prefix *str2* on the command line it matches only if the cursor is before this word.

R[*str1*,*str2*] ...
: Like *r* but using pattern matching instead.

18.5 Example

```
compctl -u -x 's[+] c[-1,-f],s[-f+]' -g '~/Mail/*(:t)' - 's[-f],c[-1,-f]' -f -- mail
```

This is to be interpreted as follows:

If the current command is `mail`, then

if ((the current word begins with `+` and the previous word is `-f`) or (the current word begins with `-f+`)), then complete the non-directory part (the `:t` glob modifier) of files in the directory '`~/Mail`'; else

if the current word begins with '`-f`' or the previous word was '`-f`', then complete any file; else

complete user names.

Concept Index

F

G

H

I

J

K

Variables Index

N

O

P

R

S

T

U

V

W

Z

Options Index

M

N

O

P

R

S

U

V

X

Z

Functions Index

Editor Functions Index

Keystroke Index

Table of Contents